FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Fault Injection in Android Applications

João Manuel Estrada Pereira Gouveia

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Ramada Paiva

June 27, 2018

Fault Injection in Android Applications

João Manuel Estrada Pereira Gouveia

Mestrado Integrado em Engenharia Informática e Computação

June 27, 2018

Abstract

The usage of mobile apps on day-to-day activities keeps increasing progressively which makes the number of applications available in leading app stores rise at an overwhelming pace. As a consequence, the quality of the software is often overlooked to maintain a fast and inexpensive delivery rhythm.

In order to avoid this scenario and ensure software quality, tests must be conducted to assess if said software is working properly and responding to input as it should. Mutation testing is a fault injection technique which stands as a valuable way of assuring the quality and effectiveness of the test cases being ran on the software itself.

Focusing on Android applications and good practices of Android development, this study aims to extend previous research work performed in the Software Engineering (SE) lab in the Faculty of Engineering of the University of Porto in which a tool was developed to test Android applications (*iMPAcT tool*). The *tool*'s approach of reverse engineering, pattern matching and testing attempts to validate if guidelines for Android development are being correctly followed.

Considering this context, the main goal of this research consists in the definition of mutation operators capable of reproducing real faults caused when the good practices of Android programming are not followed. The operators will allow to evaluate the quality of test cases defined in applications. It will also be used to evaluate the background pattern defined in the *iMPAcT tool*.

This study will be conducted by injecting the mutation operators in real applications. These applications will be tested by the *iMPAcT tool* to assess if the *tool* is able to detect the faults injected. Comparing the results produced by the *tool* regarding the original code and the mutated code, it should be possible to verify if the test cases executed by the *iMPAcT tool* are sufficient to detect the injected faults and consequently kill them. This way it will be possible to improve quality in mobile applications.

Resumo

O uso de aplicações móveis no quotidiano das pessoas continua a crescer progressivamente, o que leva o número de aplicações nas principais *app stores* a aumentar a um ritmo alucinante. Como consequência disso mesmo, a qualidade do software é menosprezada para manter ritmo rápido e barato de desenvolvimento.

De forma a evitar este cenário e a assegurar a qualidade do software, é necessário testar este mesmo software para determinar se este trabalha corretamentamente e responde a *inputs* como previsto. Testes de mutação é uma técnica de injeção de falhas que possiblita uma forma viável de a garantir a qualidade e eficácia dos testes que correm no próprio software.

Focando nas aplicações Android e nas boas práticas para o seu desenvolvimento, este estudo pretende estender trabalhos anteriores desenvolvidos no departamento de Engenharia de Software da Faculdade de Engenharia da Universidade do Porto onde foi desenvolvida uma ferramenta de testes Android (*iMPAcT tool*). Esta ferramenta utiliza *reverse engineering*, comparação de padrões e testes para validar se as linhas condutoras da programação em Android estão a ser corretamente seguidas.

Atendendo a isto, o principal objetivo desta pesquisa consiste na definição de operadores de mutação capazes de reproduzir falhas reais que acontecem quando as boas práticas de programação Android não são seguidas. Estes operadores irão permitir a avaliação da qualidade dos testes definidos nas aplicações. Também serão usados para avaliar os padrões de teste definidos na *iMPAcT tool*, nomeadamente o padrão de *background*.

Neste estudo, irão ser injetados operadores de mutação em aplicações reais. Depois, estas aplicações serão testadas pela *iMPAcT tool* de forma a verificar se a ferramenta deteta ou não as falhas injetadas. Comparando os resultados verificados nos testes ao código original e no código mutado, irá ser possível analisar se os testes executados pela *iMPAcT tool* são suficientes para detetar e matar os mutantes injetados. Desta foma será possível aumentar a qualidade das aplicações móveis.

Acknowledgements

At the end of this stage of my life, there is so much people without whom I would not be here.

First, I would like to thank my parents and my brother for always supporting me in the difficult times, whenever I thought I could not get where I am today.

I would also like to thank all my friends for these long, gruesome and amazing years of my life. I have shared so many moments with you that I will always fondly remember.

To my supervisor Prof. Ana Paiva, for having all the patient and dedication necessary to guide me throughout this process, a big thank you.

And finally, to Mariana, for all the love, understanding and most of all, for believing in me, even when I could not do it myself.

João Estrada Gouveia

“ ’Can a man still be brave if he’s afraid?’
’That is the only time a man can be brave’ ”

George R.R. Martin, A Game of Thrones

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

adb	Android Debug Bridge
APD	Activity Permission Deletion
API	Application Programming Interface
APK	Android application package
APP	Application
AUT	Application Under Test
BWD	Button Widget Deletion
DB	Database
DSL	Domain Specific Language
ECR	OnClick Event Replacement
ETR	OnTouch Event Replacement
GUI	Graphical User Interface
IPR	Intent Payload Replacement
ITR	Intent Target Replacement
MDL	Lifecycle Method Deletion
MS	Mutation Score
OS	Operating System
PBGT	Pattern Based GUI Testing
PFP	Potential Fault Profile
SDK	Software Development Kit
TP	Test Pattern
TWD	EditText Widget Deletion
UI	User Interface
UITP	User Interface Test Pattern
XML	Extensive Markup Language

Chapter 1

Introduction

The numSber of smartphone users keeps progressively increasing at an overwhelming pace. Acording to [Sta], smartphone usage has reached unprecedented numbers in 2017, with an astonishing 2.32 billion users worldwide. The forecasts regarding these numbers for 2018 are even higher, with 2.53 billion.

Android stands by far as the biggest mobile operating system in the world with more than 2 billion monthly active devices and 82 billion downloaded apps from Google Play Store during 2016 [Ver]. The tendency is for these values to keep rising, so logic would dictate that the quality of the apps uploaded to Google Play would increase as well. Unfortunately this is not the case.

With thousands of apps added every day, Google's Play Store is bloated with poor-quality software [Eng]. Google knows this and to deal with the problem is now running a set of quality analysing algorithms through the apps to rank down any badly coded ones. These quality parameters are set in the Android Vitals, which is an initiative to improve the stability and performance of Android devices by improving quality among the Android apps [Anda].

This action, in spite of getting more developers aware of good practices of Android and keeping poor quality apps off the top of apps lists, does not mitigate every problem of quality in applications [CMPF12].

1.1 Context

Regarding the current state of mobile development, Android continues to grow in popularity, which translates in a boost in the number of users. Consequently, that leads to an increase of the quantity of applications developed and available at Google's Play Store and their respective downloads. Hence, it is paramount to ensure the quality of these mobile applications.

Having in mind the constant changes and innovations in applications development, testing can turn into a troublesome task. Both companies and developers try to focus more on testing

and guaranteeing the quality of their applications since faulty behaviour may bring along serious problems like monetary losses, legal issues or brand image damage.

In spite of this investment in mobile software testing, one of the biggest setbacks is the lack of time due to the fast deployment of applications and proper methods and tools to keep up with it.

Thus, it becomes clear that it is deeply important to automate mobile application testing.

1.2 Motivation and goals

In app stores there are multiple applications that perform similar functionalities and are used for the same purpose. Knowing this and how easy it is for a user to delete an application from his phone because it is not working properly, it is imperative for companies to ensure the quality of mobile applications. In spite of the efforts of Google and other companies as well, the lack of quality is a major concern of Android applications.

When developing an application, destruction can be just as valuable as creation. Thus, an efficient test set that can break the software, catch potential bugs and attest for the correct functioning of the application is of great importance. But how can we assure that the test sets are appropriate and precise?

The *iMPAcT tool* comes to aid in this process by recurring to reverse engineering and pattern identification techniques to automate the testing of recurring behaviours (UI patterns) which are present on Android applications [CMPF12]. By using the *tool* to evaluate specific Android UI patterns, it is possible to determine the quality of code and effectiveness of the test sets of these applications.

With this study, it is intended to extend a previous research work done in this field. Real faults of Android applications resultant of not following the good practices of Android programming will be simulated by recurring to mutation operators. They will be injected in concrete applications to evaluate the test sets defined in the *iMPAcT tool*. This will reveal if these tests are able to detect and kill the mutants injected in the source code of the application under test (AUT). Also, by testing the mutants defined through the *iMPAcT tool*, they will be validated. With this, it is intended to legitimize the mutation operators defined for enabling testing of good practices in Android programming. By doing so, these mutation operators become an asset for mutation testing in Android, which is still at an early stage of development.

By injecting the mutants in selected applications, it will be possible to verify if the *iMPAcT tool*'s test set uncovers the incorrect use of UI patterns in Android applications, consequently increasing the confidence in their quality.

1.3 Structure of the document

Besides the introduction, this report contains 4 more chapters. In chapter 2, there is the description of the state of the art and similar researches, including the current state of mobile testing, mutation in Android and previous studies regarding the subject. In chapter 3, the solution to the problem

in hand is stated, the methodologies and implementation details are described. How the selection of applications was done, the mutation operators definition and their injection are some subjects addressed there. In chapter 4, the case study and results of the experiments are analysed regarding the injection of mutation operators. In chapter 5, the conclusions taken from this study and the possible future work to be done are described.

Chapter 2

State of the Art

This chapter is divided in 7 main sections: 2.1 presents Android programming and its uniqueness, in 2.2 it is described the current state of mobile testing and its approach, in 2.3 the *iMPAcT tool* is presented and how and why it is used, in 2.4 there is the definition of mutation testing and its purpose, 2.5 reviews the current state of mutation testing on Android and existing tools, 2.6 discusses the previous work done in the area and finally, 2.7 concludes the state of the art as a whole.

2.1 Android Programming

Android is an Linux-based mobile phone operating system developed by Google. The Android operating System (OS) is used to power a multitude of devices, from smartphones or watches to car stereos [Lif].

Most Android applications are written in Java-like languages. This means that, while developing for Android, the programming language used is Java. Android has its own SDK (Software Development Kit), which is a specific set of Java classes and methods for Android, i.e. dealing with click events and user interface. Therefore the programming language used to develop for Android itself is more commonly referred as "Android". Kotlin is now also an official language of Android development as well [Kot]. In spite of being interoperable with existing Android languages and runtime, it will not be used or studied in the context of this research.

Android is a component based language, meaning that each application is composed by different components. There are multiple different components, but the four main components which are the core of any Android application are *Activities*, *Services*, *Content Providers* and *Broadcast Receivers* [Andb]. The interconnection, correct programming and use of these components is what creates a well-rounded Android application. These will discussed in depth next.

2.1.1 Activities

An activity is where the user interacts with the application, representing a single screen with a user interface. Each one of the activities of an application are independent from the others. This is helpful when it is necessary to have a different application or the system communicating with any of these activities without having the need to communicate with the whole application. Every activity can have different states: *Created*, *Started*, *Resumed*, *Paused*, *Stopped* or *Destroyed* [Act]. The state of the application is controlled by callback methods which are fired to initiate each one of the states that compose an activity lifecycle, as seen in figure 2.1:

- **onCreate()** — It is fired when the system first creates the activity and the activity's state is set to *Created*. Here is where the basic setup logic that should only occur once in the activity happens. When this initialization is finalized, the activity in set to the *Started* state [Act].

- **onStart()** — When the activity enters the *Started* state, the system invokes this callback. When this happens, the application becomes visible to the user and becomes interactive. For example, this method initializes the code that maintains the User Interface (UI). This callback might also register a Broadcast Receiver that verifies changes that are reflected in the UI. As soon as this callback ends, which is usually very quickly, the activity jumps to the *Resumed* state [Act].

- **onResume()** — This method is invoked when the activity is on the *Resumed* state. It is here where the app interacts with the user. It will stay in this state until something happens to take the focus from the app, like receiving a phone call or navigating to another application. When such an interruptive event occurs, the activity is set to the *Paused* state [Act].

- **onPause()** — The system invokes this callback when the activity is in the *Paused* state. This means that the user is leaving the application, although it does not necessarily mean that the activity is being destroyed. The method pauses operations such as animations, music playback or others that should not continue while the app is in the *Paused* state (running in the background) [Act].

- **onStop()** — This callback is invoked when the activity is no longer visible to the user and so it enters the *Stopped* state. The app should release almost all the resources that are not needed when the user is not using it and also resources that might leak memory [Act].

- **onDestroy()** — This method is fired before the activity is destroyed. It is the final call that the activity receives. If there are any resources that are yet to be released, they should be so here [Act].

- **onRestart()** — This method is called after the activity is stopped before being started again. It is followed by the *onStart()* callback [Act].

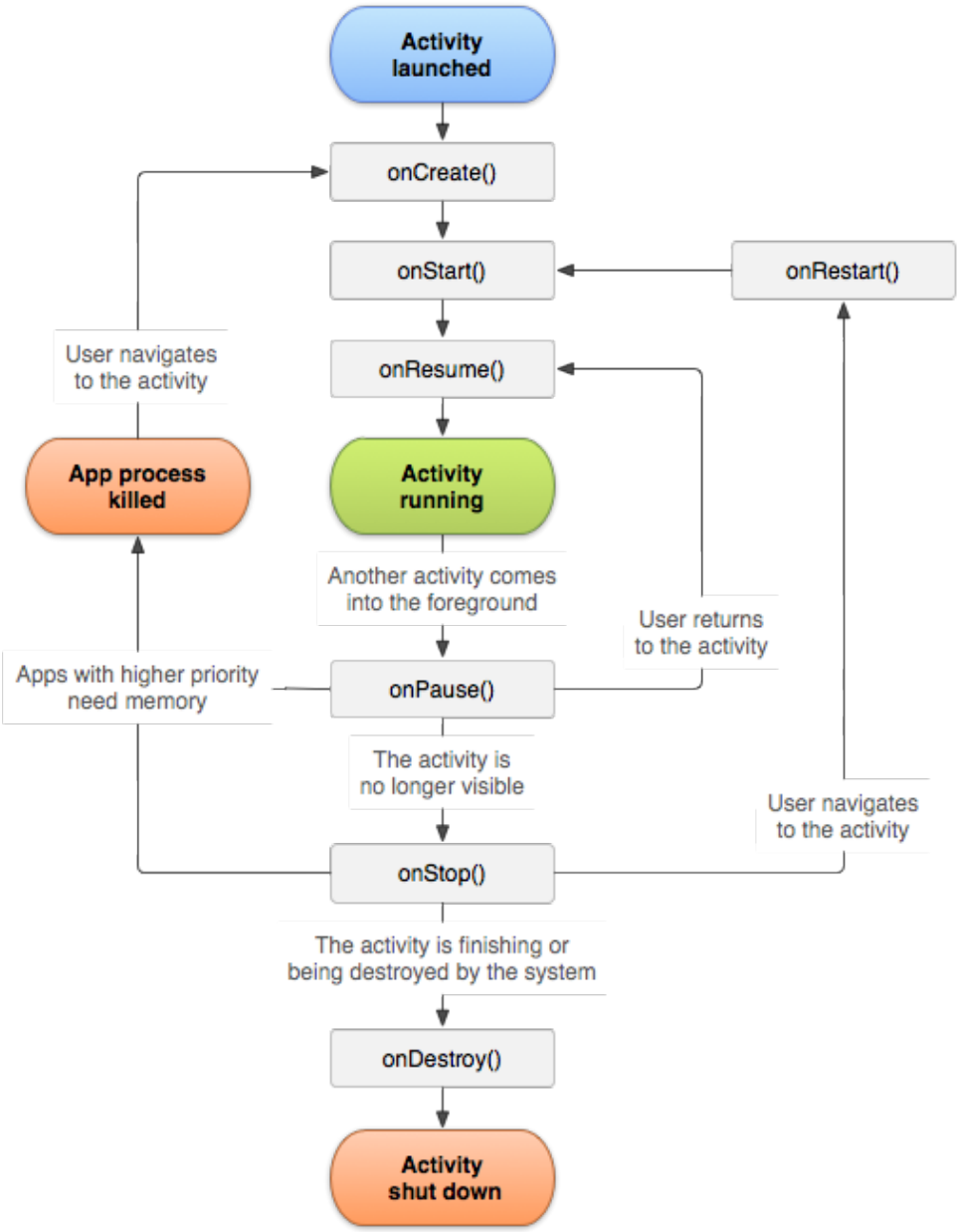


Figure 2.1: A simplified illustration of the activity lifecycle [Act]

All these methods exist by nature in every activity of an application but it is possible to override them. This is usually done if there are some specific actions that need to be addressed in each of the states of the activity.

It is paramount to understand what must be done in each one of these states and how can one influence the other. The misunderstanding of the lifecycle of an activity may lead loss of information and memory leaks, that may cause the application to crash.

2.1.2 Services

A Service can be used as an entry point in case it is required to keep the application running in the background to perform long-running operations or work for remote processes. A service does not have a UI. They are divided in two different semantics [Ande]:

- Started services — responsible for telling the system to keep the app running until the work is complete.

- Bound services — allow another app or process to make use of the service. This is done so the service can provide an API of some sort to another process.

An example of a service is a Music Player that keeps playing in the background in spite of another app being on the screen.

If badly implemented, services may lead to unnecessary memory usage and strange behaviours in other applications that exist in the same device. That is why they need to be dealt carefully in order not to harm the entire system.

2.1.3 Broadcast Receivers

A Broadcast Receiver allows the app to respond to system-wide broadcast announcements. This way the system can deliver broadcast events to apps that are not currently running. They can also be initiated by the system. As an example, if it is low on battery, the system can initiate a Broadcast Receiver to warn running apps about it or even apps that consume too much battery while running that should not do so under this condition. They can also be initiated by applications. An app can schedule an alarm to post a notification. If the app delivers the alarm to a Broadcast Receiver, there is no need for it to keep running [Andc].

Although Broadcast Receivers do not have a UI, they may create a status bar notification to alert the user when a broadcast event occurs.

2.1.4 Content Providers

A Content Provider is used to manage app data that needs to be stored in a persistent storage location that the app can access, either a database, locally in the file system or in the web. Through the content provider, other apps can query or modify the data if the content provider allows it [Andd].

As described in this section, Android programming can be challenging and different from other languages. It is important to understand how each component interacts with one another in order to create cohesive and clean code. Only by doing so it is possible to develop quality Android applications that jeopardize the system as a whole.

2.2 Testing Mobile Applications

Mobile application testing is when the applications that run on mobile devices and their functionalities are tested. There is an element to consider that is very important and can affect the way the app is tested. An app can be either a native app, web app or hybrid app. There are a few differences between these types of app. Native apps are single platform while web apps are cross-platform. The way to test each one of these types of applications may vary and have distinct processes to do so.

When testing a mobile application, there are a multitude of aspects that need to be taken into account [Sof]. There are several things that can influence mobile testing:

- Different mobile operating systems (Android, Windows, IOS, among others)
- Diversity of mobile devices (Apple, Samsung, Nokia, Motorola, etc)
- Wide range of characteristics in mobile devices (different screen size, hardware configurations, etc)
- Different mobile network signal (Wifi, 4G, 3G, etc)

Moreover, when compared to Desktop development there are details that make mobile applications more constrained. The smaller screens, the limited memory and processing resources and the distinct network connections are just a few factors that show the differences between these two platforms.

Besides this, mobile development has different features like the input being done from a touch screen rather than a mouse and keyboard, the possibility to rotate the screen, the multiple sensors (Bluetooth, NFC, gyroscope, among others) which are context providers that may give a huge amount of inputs. This makes mobile development (and consequently its testing) fairly complex.

It is possible to test a mobile application using emulators (that simulate a desired software environment) or on a real device. Both these approaches have their advantages and drawbacks depending on the app's lifecycle.

Emulators are great in early stages because they can test the application in multiple devices with the app running unmodified, being a fairly easy and inexpensive solution to study the behaviour of the app. But they can be slow, may only support some OS versions and not generate the same real time network connections or real device characteristics [PMVS14].

Testing on real devices, on the other hand, is crucial to ensure that the app is ready to hit the market and be used regularly. This is done using a real environment with real hardware and software, enables testing interoperability and battery drainage issues much easier, and so forth [Sau].

To better understand how to test a mobile application, it is paramount to know the different types of testing. In [MdFE12] the matter is divided in five unique aspects to take into account:

- **Performance and reliability testing** — Performance and reliability of an application are deeply correlated with the available resources, connectivity quality, among other elements, of a mobile device.

- **Memory and energy testing** — Mobile devices have limited resources, not only battery wise but also memory so it is paramount to avoid memory leaks and abusive consumption of battery by the applications on the smartphones.

- **Security testing** — Since mobile devices are constantly connecting to different networks and receiving a multitude of different inputs, it becomes clear that security testing is of particular relevance. These devices are data centric and carry private or sensitive information like bank account details, passwords and other relevant data. Smartphones and other devices are regularly vulnerable to various threats and attacks so it is important to protect them against leaks of these private and sensitive data. In [WA15] it is stated that there is still a deficit of approaches to detect leakage, especially when this data is encrypted.

- **GUI testing** — Graphical User Interface (GUI) testing regards the display of the application to the user and it is where the user interacts with it. It is critical to test the GUI since if it is not working properly, the user will not be able to use and interact with the application. [MdFE12] suggests to "automatically execute scripts that are captured during the user interaction and replayed and modified even on different devices". This is the idea behind the *iMPAcT tool*, which is explored next in 2.3. Another tool is the A2T2 (Android Automatic Testing Tool) which uses "a GUI crawling based technique for crash testing and regression testing of Android applications" [AFT11].

- **Product line testing** — Especially when talking about Android, there are a lot of different hardware/software combinations (different producers with different hardware using different Android versions). This means that applications need to work properly in a large number of different devices.

2.2.1 Existing Tools

Being Android programming one of the focal points of this research, the tools that are evaluated are for testing in Android environment. In spite of these tool's not being applicable to this study, understanding their functionalities can be an asset to the goal of this research.

2.2.1.1 Pattern Base GUI Testing (PBGT)

PGBT introduces a methodology to "sample the input space using 'UI Test Patterns'" [MPM13]. This kind of testing automates and systematizes the GUI testing process, which is often overlooked. GUIs are fundamental in interacting with programs and this empirical study is effective in revealing faults in the GUI of said programs. The PGBT recurs to Model-Based Testing to generate test cases. These models are "written in a Domain Specific Language called PARADIGM and are composed by User Interface Test Patterns describing the testing goals" [NP14]

PARADIGM is a domain specific language language (DSL). DSLs are "high-level languages exclusively tailored to specific tasks" [MP14]. Since the PBGT model requires a distinct way to

handle GUI models providing UI Test Patterns that can be configured for different testing implementations, there was the necessity to develop PARADIGM to enable reusability of existing elements or extension of the same. PARADIGM also allows building a model to describe test goals instead of describing the expected behaviour [MP14].

PGBT creates a model based in testing goals (test patterns). These are configured with test data and later the test suites are generated with this information. The test suites are then automatically executed through the GUI of the application being tested. At last the PGBT produces a report or log with the results of the tests executed against the AUT.

With its fundamentals in PBGT, some pattern based tools were later developed by extending PBGT's PARADIGM language:

- Pattern Based Usability Testing — performs automatic usability tests on web interfaces in order to understand recurrent usablity issues over different web applications [DP17].
- Pattern Based Web Security Testing — focuses on implementing test strategies to analyse if the applications are vulnerable to aspects regarding the patterns defined ("Account Lockout" and "Authentication Enforcer") [JMAP18].
- Pattern Based GUI Testing for Mobile Applications — reuses most of the PBGT to test mobile applications by the changing the mapping and interaction strategy with the application under test [CPN14].

In [MPNM17] two different case studies were conducted in the PBGT. The first one was carried out in order to assess the PBGT failure detection capabilities by understanding if testers were able to find failures the system tested and how much time they spent doing it. In the second one the PBGT's generation strategies of test cases were analysed, by comparing it with random testing and manual model-based testing. Both case studies proved the PBGT tool's ability and effectiveness to detect failures.

2.2.1.2 Android Studio

Android Studio offers a simple way to set up a JUnit test to run on the local JVM (Java Virtual Machine). It is also possible to integrate the application with several testing frameworks like Mockito, Espresso or UI Automator [Andf].

2.2.1.3 Mockito

Mockito is an open source testing framework for Java. It is used for Test-Driven Development (TDD) or Behaviour-Driven Development (BDD) [Moc]. It is a mocking framework that allows developers to verify the system under test without establishing any expectations beforehand. It attempts to eliminate the expect-run-verify pattern. One of the the criticisms is that, by using mock objects (simulated objects that mimic the behaviour of real objects in controlled ways), Mockito does a tight coupling of the test code and the system under test.

2.2.1.4 Espresso

Espresso is a UI testing framework. It is intended to test a single application but can also be used to test across applications. It has four main components [Exp] used to write concise and reliable Android UI tests:

- **Espresso** — Entry point to interactions with view. Also exposes APIs that are not necessarily tied to any view.
- **ViewMatchers** — Allows to find a view in the current view hierarchy.
- **ViewActions** — Allows to perform actions on the views.
- **ViewAssertions** — Allows to assert state of a view.

2.2.1.5 UI Automator

"The UI Automator testing framework provides a set of APIs to build UI tests that perform interactions on user apps and system apps. " [UIA]

It makes it possible to perform operations such as opening the Settings menu or the app launcher in a test device. The framework allows writing black box-style automated tests without having them rely on internal implementation details of the target app.

It's key features are:

- **UI Automator Viewer** — provides a GUI to check the UI components that are being currently displayed on the mobile device. It can be used to verify the layout hierarchy and the properties of the UI components that are visible on the screen.
- An API to retrieve state information and to perform operations on the target device where the application is running.
- **UI Automator APIs** — support cross-app UI testing.

2.2.1.6 Monkey and monkeyrunner

"The Monkey is a program that runs on your emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events" [Exe]. It is used to stress-test applications in development in order to assess if their performance is adequate under extreme and unfavourable conditions, like underclocking, overclocking, heavy network traffic, among others [Tec].

"The monkeyrunner tool provides an API for writing programs that control an Android device or emulator from outside of Android code" [Mon]. Its primary purpose is to test applications and devices on a functional level by installing an APK or test package, running it and sending keystrokes to the application's UI and saving screenshots of various states of the application.

2.2.1.7 Robotium

“Robotium is an Android test automation framework that has full support for native and hybrid applications” [Rob]. It is used for writing automatic black box UI tests for Android applications. It can handle multiple Android activities automatically, access and modify the mobile device’s sensors and service’s state and does not require access to the application’s source code. On the other hand, it cannot read the content on display on the screen of the device and it can only modify services and sensors if it has the needed permissions.

2.2.1.8 Appium

"Appium aims to automate any mobile app from any language and any test framework, with full access to back-end APIs and DBs from test code" [App]. Appium does not require access to the applications source code, can test native, hybrid and web applications and can modify services and servers states. Besides this, it is fairly unstable, sometimes creating unexpected errors and does not return the screen content.

The tools presented allow to understand more about the complexity of Android as a programming language and Android testing. There are multiple tools that enable testing of different areas of Android programming since it is vast and there is still huge room for improvement in the field.

2.3 The iMPAcT tool

"The *iMPAcT tool* automates the testing of recurring behaviours (UI patterns) present on Android mobile applications" [MP16]. The *tool* explores the application under test (AUT) to find UI patterns and tests them against the strategies associated with each pattern, which are implemented in the tool.

This tool emerged as a follow up of the PBGT but specific to mobile applications and in an attempt to test particular behaviour in mobile applications. Also, this tool tries to automate the testing process even further as a extension of what PBGT already does.

A pattern can be defined as a recurring solution for a recurring problem. Translating this into mobile testing, an UI pattern is a representation of a specific behaviour in a mobile application. This means that if an application implements a certain feature, for example a login, it needs to have specific elements, like a user name input, a password input and a login button. This way it is possible to verify the existence (or the lack of) these elements to check if the pattern is correctly implemented.

As stated in [MP15], UI test patterns follow a simple template to uniform their description:

- Pattern Name — unique identifier of the pattern
- Context — conditions in which the problem is verified
- Problem — problem addressed by the pattern

- Forces — details to consider when choosing a solution for the problem in hand
- Solution — description of the actual pattern
- Consequences — positive and negative outcomes of the implementation of the solution
- Application Candidates — Real conditions in which the pattern can be applied

One of the most important parts of the *iMPAcT tool*'s approach is the pattern catalogue defined in it, which contains the definition of each UI pattern and corresponding test case [Mor17]. As described in [MP16] and [Fer17], there are several patterns implemented in the *tool*, which are:

- **Side or Navigation Drawer pattern** — The different screens and hierarchy of an application includes the *Side Drawer* (or Navigation Drawer) UI pattern, which is the main menu of the application, displayed on the left side of the screen. The *Side Drawer* is hidden most of the time and will appear on the screen by swiping the screen from the left edge of the screen to the middle, or by clicking the menu icon which is on display. This pattern tests if the *Side Drawer*, when opened, occupies the full height of the screen [Mor17].

- **Orientation pattern** — A mobile device has two possible orientations, landscape (when the width of the display is bigger than the height) and portrait (when the height of the display is bigger than the width). After the rotation of the screen, there is an update on the devices display. Posterior to this change, it is imperative that the main components of the previous display are still present, meaning that no information is lost [Mor17].

  This pattern tests if:

  1. Data is not lost when the screen rotates
  2. UI main components do not disappear after the rotation

  In order to do so, the *iMPAcT tool* matches the elements of the screen prior to the rotation with the ones post-rotation, one by one, according to all their property except their position, to evaluate if the data contained in all of them remains unchanged. The *tool* also verifies if the main widgets of the screen before the rotation are still present afterwards, although their content may be different. The main components considered are the *Action Bar*, the *Side Drawer*, *Radio Groups* and *ListViews*.

- **Resources Dependency pattern** — Several applications need the use of external resources, like GPS or Wifi, to work properly. It is paramount to check if these resources are available or not and if the application does not crash in their absence or if some resource is suddenly unavailable [Mor17].

  To achieve this, the *tool* verifies if the application is using a certain resource, and if so, turns said resource off. Then, the application's state is checked, to attest whether the unavailability of the resource caused some error or even crashed the application.

- **Tab pattern** — A tab makes the navigation between different views of an application easier. The *iMPAcT tool* tests this pattern against some guidelines to ensure its correct implementation. After detecting the existence of a tab, the *tool* tests whether the pattern is correctly implemented by verifying if [Mor17]:

  — there is only one set of tabs per activity

  — the tabs are correctly positioned on the upper part of the screen

  — the horizontal swipe motion on the screen changes the selected tab and nothing else

- **Background pattern** — When using an application and the home button is pressed, the screen should return to the home menu, leaving the application running in the background and saving its current state. This action does not make the application to fully exit. If the user wants to go back to that application, he should either click on the app button on the smartphone or open the recent apps menu and look for it. By opening the app, the screen presented should be the same and the application should be in the same state as before [Fer17].

  To test this pattern, the *tool* clicks in the home button, which sends the AUT to the background. Then, it opens the recent apps menu and opens the application again. Afterwards, the screen from before closing the app and the screen after reopening it are compared. The goal of this test is to check if the app goes to background when the home button is pressed and if, when bringing it back to the foreground, its state remains the same. This pattern exists in an app by nature, so the *iMPAcT tool* does not formally check its existence.

- **Action Bar pattern** — In an Android application, the Action Bar is used to various purposes, like locating the user within the app, giving access to important actions and supporting navigation and changes in the view. It also gives flow between all Android apps, which helps the user to quickly understand how to use it. The Action Bar must be placed at the top of the screen and presents the apps title and a floating menu, the first on the left side and second on the right side. The floating menu must contain the most important actions of the app. The Action Bar should also includes on the leftmost side an Up carret when not on the main screen or a button to open the side drawer, when it exists [Fer17].

  The goal of this test is to verify the correct positioning of the Action Bar on the screen and if its components follows the rules stated. This pattern exists in an app by nature, so the *iMPAcT tool* does not formally check its existence.

- **Up pattern** — According to the good practices of Android programming, every screen in the application that is not the main one should offer the possibility to go to the one that is logically its parent in the hierarchy. This is done by pressing the Up button, present in the Action Bar. This is done so the user can easily retrace his action an follow his path back to the main screen [Fer17].

  This pattern tests if :

— in every screen of the application (except the main screen), there exists an Action Bar and, in the cases where the Action Bar does not contain a Side Drawer, it contains an Up button.

— in the case there is an Up button, when it is clicked, it sends the application to the current screen's logical parent in the hierarchy.

- **Back pattern** — In all Android devices, there is a Back button which is provided to correctly use the back navigation, in spite of current state of the app. In default, the system will recur to the *back stack* to process the back navigation (with the exception of some cases like switching between fragments) [Fer17].

  The goal of this pattern is to verify the correct use of the Back button by the application. To do this, the *tool*:

  — verifies if the AUT uses the Back button provided and not a personalized one.

  — checks if the AUT is not in the initial screen (because if it is there is no point in testing this pattern).

  — checks if, when the Back button is clicked, the application changes to its previous visited screen.

While not thurly extensive, this catalogue is "based on the guidelines provided by Android on how to design applications and on how to test them" [Mor17].

The *iMPAcT tool*s approach consists "in continuously exploring the AUT (Explorer) while identifying the presence of UI Patterns in the application (matches) and testing them (Tester)" [MP16]. This is visible in figure 2.2.

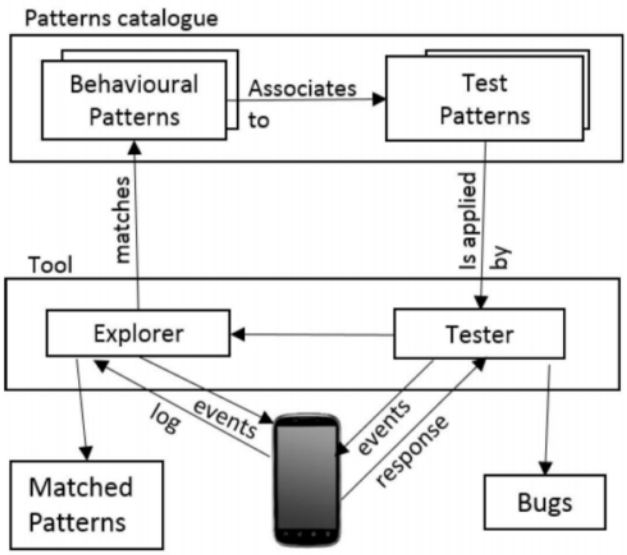


Figure 2.2: Block Diagram of the Architecture of the *iMPAcT tool* [MP16]

The *iMPAcT tool* approach and execution on each application divides itself in three phases:

1. **Exploration** — This phase regards the exploration of the AUT and the screen on display searching for possible events to be fired. One of the events found is randomly selected and it is fired.

2. **Pattern Matching** (or reverse engineering) — The *tool* verifies if any of the UI patterns defined in the catalogue is present. If all the pre-conditions for testing of a pattern exist in the application, the pattern is considered found.

3. **Testing** — If a certain UI pattern was identified in the previous phase, the corresponding tests are retrieved from the catalogue, in order to apply these test strategies. If the tests fail, this means that the pattern is not correctly implemented. If they hold, the pattern is considered correctly implemented and a report is produced. If there is no pattern identified in the Pattern Matching phase, then this stage is skipped. After all tests are executed, the exploration of the application is resumed.

"At the end of the exploration two main artefacts are produced: the report of the exploration and a model of the behaviour observed during the exploration" [Mor17]. The report has a log of the exploration done on the AUT and the result of the tests to which the application was submitted.

When executing the *tool*, there are four exploration algorithms to choose from, which are [Mor17]:

- **Execute once** — Each one of the possible events is only fired once, meaning that when an event is fired it is no longer valid.

- **Priority to not executed** — The events that are yet to be fired have priority over the ones that have already been fired. When every event has been fired, the algorithm chooses the one which is more likely to lead to a different untested screen. To avoid loops each event can only be fired a previously set number of times.

- **Priority to not executed and list items** — similar to the previous algorithm, with the addition that events associated with lists have also a higher priority of being fired since they can lead to further exploring a screen before a certain event changes it.

- **No restrictions** — Every event will be fired.

By choosing a different exploration algorithm, it is possible to land in different screens of the application, which enables to test distinct parts of it.

The *iMPAcT tool* proves to be a valuable testing framework due to its characteristics. The fact that it does not require access to the source code of applications (because of its screenshot technique), the multiple patterns implemented and the different exploration algorithms makes it a versatile tool for Android testing.

2.4 Mutation Testing

Mutation testing is a fault-based software testing technique. It is used to assess the quality of test cases by injecting mutants (changing certain statements in the source code) and analysing if the test cases are able to find these errors. It is a type of white box testing technique mainly used for *Unit testing*.

"Fault-based testing strategies are based on the notion of testing for specific kinds of faults. Fault-based testing strategies succeed because programmers tend to make certain types of errors that can be well defined" [Off92].

"The general principle underlying Mutation Testing work is that the faults used by Mutation Testing represent the mistakes that programmers often make" [JH11]. These faults are small, from simple sintatic changes, like switching to mathmatic operators, to deletion of statements. Each of these faults is called a *mutant operator* [Rib17].

Mutation testing's goal is to generate all faults that could be present in an application. Since the number of potential faults in an application can be a huge number, mutation testing focus on a subset of these faults, the ones which are closer to the original program [JH11].

Acording to [JH11] and [Off89], the fault set is commonly restricted by two principles or hypothesis:

- *The Competent Programmer* — This hypothesis states that programmers are competent, which means they develop applications that are close to the correct version. Hence, the only faults introduced in the programs are simple faults, like small syntactical errors.
- *The Coupling Effect* — This hypothesis states that test cases that can detect simple faults will also be able to detect the majority of the more complex ones.

2.4.1 The Process of Mutation Analysis

Mutation testing and the introduction of mutants in the source code follows a set of steps, which are [Rib17]:

- The test set is ran against the source code. If there is some failure it must be corrected before proceeding.
- The faults are introduced in the source code. Each alteration done in the code is called a mutant. These are created according to the selected mutation operations.
- The test set is now ran against the mutated code. The results are compared against the ones obtained in the first step.
- If the results are different, this means that the mutants were killed and the test set is considered capable since it detected the mutant injected.
- Otherwise, if the results have the same output, the test case was not able to detect and kill the mutant, hence it is still alive.

- If there is no test case capable of killing the mutant, it is called an equivalent mutant. These do not modify the meaning of the original program and should be discarded.

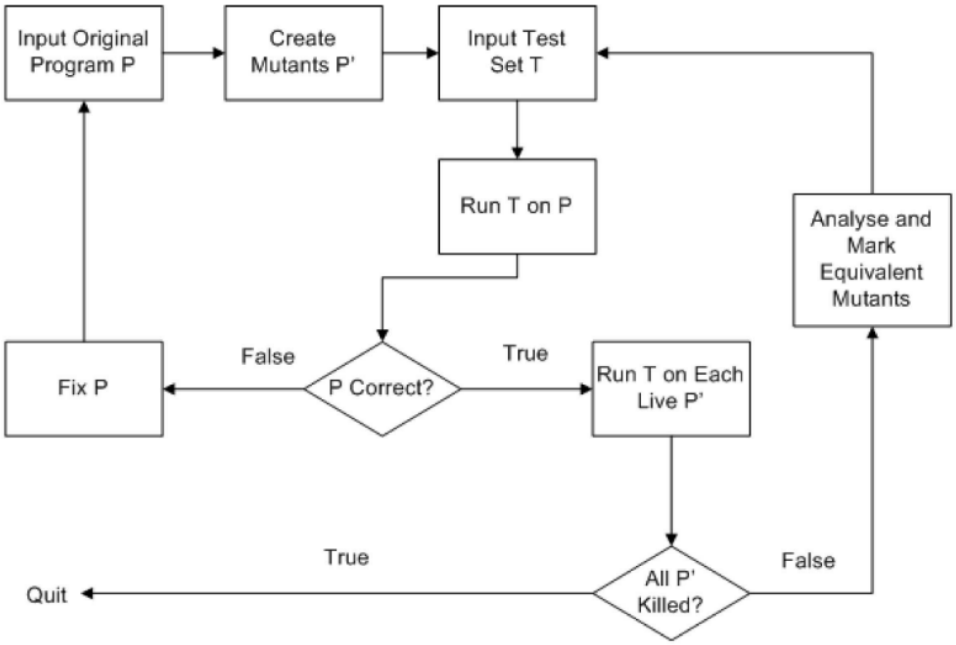


Figure 2.3: Generic process of mutation analysis, from [JH11]

In figure 2.3, there is the representation of the mutation testing process for a single mutation operator (mutation operators are discussed in section 2.5). "For a program *p*, a set of faulty programs *p'*, is generated by a few simple syntatic changes to the original program *p*." [JH11]

This process can be slow if there are a large set of mutants created. After this process, it is possible to access the effectiveness of a test set to detect the faults injected (kill the mutants). This is achieved by calculating the *Mutation Adequacy Score* (or just *Mutatcroion Score*), using the formula in 2.4.1.

Mutation Score (MS) = Mutants Killed / (Total Mutants - Equivalent Mutants)

The value retrieved from this equation is between 0 and 1. If the value is 1, all the mutants were killed by the test suite (optimal case). If the value is 0 no mutant was killed by the test suite (worst case). This means the closer the value is to 1, the better the test suites are able to detect and kill the mutants injected.

As described in [DOA14], test requirements have a lot of duplication, in a way that test suits tend to test the same features or output the same results. This means that hundreds of mutants can be killed by a few tests. That is why it is necessary to design mutation operators carefully, to make mutation testing more cost-efficient. If correctly defined and implemented, it can lead to efficient and powerful tests. Otherwise it will probably result in ineffective and redundant tests.

2.5 Mutation testing on Android

Mutation testing on Android is not a very well explored and documented process yet but there has been already some development in the area. In [DMAO15], there is a definition of mutation operators to use in Android. "Mutation operators have been created for many different languages, including C, Java, and Fortran[...]. Mutation operators for Android apps focus on the novel features of Android, including the manifest file, activities, services, etc. [DOA14].

2.5.1 Mutation Operators

"Mutation analysis relies on mutation operators, which are syntactic rules for changing the program or artifact" [DMAO15]. Mutation operators are divided in Android in four diferent categories: *Intent Mutation Operators*, *Event Handler Mutation Operators*, *Activity Lifecycle Mutation Operators* and *XML Mutation Operators* [DMAO15].

2.5.1.1 Intent Mutation Operators

An Intent can be described as an abstraction of an operation which will be performed on an Android component. Intents are usually used to lauch activities and to send or receive data between activities. [DMAO15] divides these type of mutation operators in two different categories, which are:

- *Intent Payload Replacement (IPR)* — it is a mutation operator that changes the payload attribute (data sent in an Intent) to a default value, to verify if the value passed by an Intent object is correct or not.
- *Intent Target Replacement (ITR)* — it is a mutation operator that replaces the target of each Intent with all classes within the same package, in order to verify if the target activity or service is successfully lauched after the Intents execution.

2.5.1.2 Event Handler Mutation Operators

Since Android is an event-based language, it uses event handlers to recognize and respond to events. In [DMAO15], there is the definition of two mutation operators for event handlers:

- *OnClick Event Replacement (ECR)* — searches and stores all event handlers that respond to *OnClick* methods present on the class being tested and procedes to replace each handler with every other handler that is compatible. A mutant of this type can only be killed by testing the *OnClick* event at least once.
- *OnTouch Event Replacement (ETR)* — works similar to the ECR, but with the diference that it changes the *OnTouch* handlers instead of the *OnClick* ones. In order to kill this type of mutant, the *OnTouch* event must be tested at least once.

2.5.1.3 Activity Lifecycle Mutation Operators

The mutant operator defined by [DMAO15] in this category is *Lifecycle Method Deletion (MDL)*. It deletes the methods that override transitions between activities life cycle states. This forces the tester to verify if the app is in the correct and expected state.

2.5.1.4 XML Mutation Operators

Android recurs to XML, not only in the manifest file, but also to define user interfaces and the default launch activity, among other uses. In [DMAO15], there are three mutation operators defined in this category, which do not modify executable code, but static XML. They are:

- *Button Widget Deletion (BWD)* — searches the XML layout file of the activity's UI and deletes, one by one, the buttons that are present. To kill this type of mutant, it is necessary to verify if every button is displayed correctly.
- *EditText Widget Deletion (TWD)* — goes through the XML layout file of the activity's UI and deletes every EditText widget present, one at a time. To successfully kill a mutant of this kind, there needs to be a test that uses each one of the EditText widgets present in the activity.
- *Activity Permission Deletion (APD)* — searches the application's manifest file (where all the permissions of the application are granted, which must be agreed by the user when installing the app) and deletes the permissions from this file, one at a time. The way to kill this type of mutant is to test a functionality that requires said permission. If the mutant cannot be killed by any test (equivalent mutant), it means that the permission was unnecessary for the app to work properly. "This is a security vulnerability that can threaten the system beyond the app" [DMAO15].

Having in mind the different mutation operators described and the fact that they are all specific to Android, it is clear that Android as a component-based language can be complex and that each mutation operator tests a different part of the test case regarding distinct components of Android.

2.5.2 Existing tools

Since it is not a very well researched area, there are not many tools to explore regarding this theme. The tools presented are used for the automation of mutation injection.

2.5.2.1 MuDroid

In [Wei], it is presented a tool called *MuDroid* which is a mutation testing tool for Android.

MuDroid's mutation operators were defined having in mind that Android is Java-like hence "most operators designed for Java could be adopted by Android as well." [Wei] This tool has six

mutation operators implemented and they are more general mutation operators and not specific to Android.

Acording to [Wei], *MuDroid* has three main phases which are:

- **Mutant Generator** — This first phase is where the APK mutants are generated from the application file which is being tested. Each APK generated has an unique seeded fault. The number of APK mutants are reduced using mutant selection rules. These APKs are then sent to the *Interaction Simulator*.
- **Interaction Simulator** — This phase produces screenshots from the APK mutants it receives. It simulates user behaviour by triggering events through adb (Android Debug Bridge). The recorded screenshots are sent to the *Result Analyser*.
- **Result Analyser** — This is the last phase of the *MuDroid* testing. Here, the screenshots received are processed to check if the mutants where killed. If the screenshot captured during testing is different from the original, the mutant is marked as killed. It also generates a report with details about equivalent mutants present in the sample APKs.

This tool does not need the application's source code to insert mutants, which is a great asset. However, it deals with Smali code (more readable than binary bytecode), which makes it slower. Also, it only implements six general mutation operators, which is not ideal since Android has a lot of complex and distinct features.

2.5.2.2 MDroid+

MDroid+ was developed in order to automatically seed mutations in Android apps. To validate the taxonomy and MDroid+ tool itself, it was conducted a comparative study with Java mutation tools. The taxonomy was derived by analysing bug reports and bug fixes of open source apps, Android-related Stack Overflow (SO) discussions and more [LVBT+17].

This enabled the extraction of 38 mutation operators, divided in 10 categories: Activity/Intents, Android Programming, Back-End Services, Connectivity, Data, Database, General Programming, GUI, I/O and Non-Functional Requirements [DMAO].

MDroid+ statically analyses the targeted mobile app, looking for locations where the operators defined in the tool can be implemented, in order to generate a Potential Fault Profile (PFP). MDroid+ will then generate a mutant for each location in the PFP. Therefore, given a location entry in the PFP, MDroid+ automatically detects the corresponding mutation operator and applies the mutation in the source code [DMAO].

It is possible to verify that *MDroid+* is a much more complex and complete tool that can generate mutants across many categories, especially Android programming which *MuDroid* does not. This is very important given the uniqueness of Android programming when compared with Java for example.

2.6 Previous Work

In [Rib17], the same problem was proposed and researched. The author pointed the same issues regarding the state of Android programming and the lack of effort to follow its good practices by programmers.

The same formal steps designed for this research were taken (described in chapter 3).

For the choosing of the applications several criteria where held in consideration, displayed in figure 2.4.

	Criteria
Be available on the Google Store	application has to be available in the Google store
Be available in Portugal	the application has to be available in Portugal to access its information
Have the source code available	access to the source code is needed to analyse the code and to insert the mutants
Have a Google store rating >= 3.5	to analyse only the applications that are have some degree of quality according to its users
Have a number of ratings >= 100	to insure that the rating of the application is a representation of the experience of several users and not only a small group
Use gradle	to simplify the build of the application
Be Android Native	because the aim of this work is to define mutation operators that are specific to Android
Have a GUI	the *iMPAcT tool* tests the GUI so it is important that the applications have one and are more than just a launcher or a keyboard
Be in an Western European Language	to facilitate the understanding of the UI of the application

Figure 2.4: Criteria for choosing applications, from [Rib17]

The criteria selected was designed having in mind that:

- the source code of the applications must be available in order to inject the mutations.
- the application should be recognized as being of quality.
- the app should use gradle to simplify the build.
- to be tested by the *iMPAcT tool*, the application must have a GUI.

There were chosen 167 applications, which were tested twice using the *iMPAcT tool* (once with the original code, once with the mutated code) [Rib17].

The author took into consideration certain aspects regarding the good practices of Android programming when defining the mutation operators used for the research done. These operators affected the behaviour of Android native elements/actions like Side or Navigation Drawer present in each activity, the change of orientation in the screen upon rotating the device and the Tab present in each activity.

The definition of these mutation operators allowed the evaluation of the test sets of the *iMPAcT tool*, by injecting these operators in Android applications and running the mutant apps against the patterns defined there.

The patterns tested were *Side or Navigation Drawer pattern*, *Orientation pattern* and *Tab pattern*.

At that time there was one more pattern with test strategies defined in the *tool* which was the *Resource Dependency pattern* but since no operator defined affected the behaviour it tests in applications, this pattern could not be inspected.

The automation of mutation operators injection was achieved by developing a Java application, which assumes that the code of the target Android application has no compiling errors and uses gradle. Not every mutant operator was automated (only those related with the *Tab pattern* and the *Side Drawer* pattern were) [Rib17]. According to [Rib17], two problems were found in the *iMPAcT tool*: the *tool* is not able to detect errors in the search widget and cannot reach all the possible screens of an application. This can lead to undetected errors if the error in a screen never tested by the test suite.

As a final remark, the author also points that the tool developed to automate the mutation operators injection has some hardcoded values and it would be a great asset to develop a GUI to the tool and automate some more operators to facilitate the work necessary for mutation injection.

2.7 Conclusions

The importance of testing the applications we produce has never been greater. With so many apps with the same purpose on the market, it is paramount to assure our applications have the most potential to succeed.

The *iMPAcT tool* provides a good way of testing Android applications. Its way of testing is ingenious due to the screenshot technique, which makes it independent of source code access of applications. This makes it easier to test the GUI of Android applications. Besides this, it can only test well defined Android behaviours, hence it should be used together with other testing tools.

Mutation testing in Android is still an unexplored area. As far as we know, the only tools available for automated mutation testing in Android are *MuDroid* and *MDroid+*. *MDroid+* implements specific mutation operators for Android programming, which the *MuDroid* does not, and those are very important in the context of this problem.

The mutation operators defined in [Rib17] do not generate mutant apps capable of testing every problem regarding the lack of good Android programming practices, so it is necessary to extend it in order to improve the efficiency of the test sets of the applications.

With this in mind, the contribution of this research will be that by assessing the correctness and effectiveness of the pattern tests of the *iMPAcT tool* using mutation operators of many categories, this area will be further investigated, improving software quality among Android applications.

Chapter 3

Process for the Definition of Android Mutators

In order to develop coherent and complete Android mutation operators, it was necessary to conduct a series of planned actions to achieve the best results possible. The scheme of activities represented in 3.1 is an overview of the methodologies followed during this research.

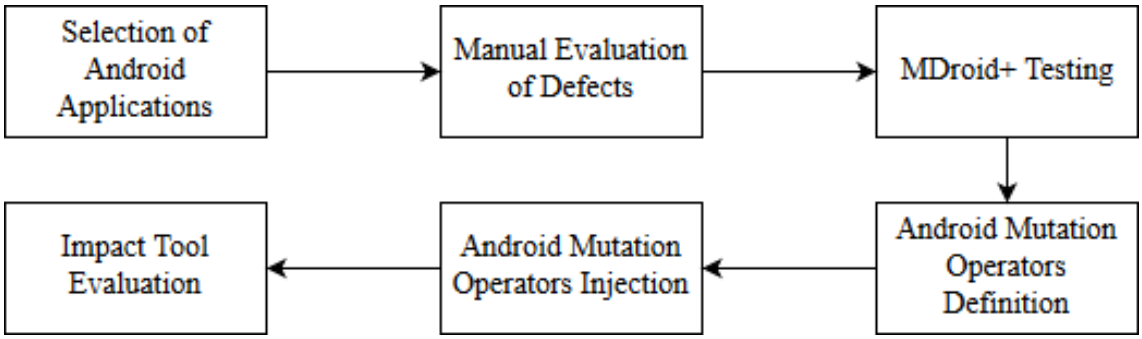


Figure 3.1: Activities followed during the process of the study

3.1 Selection of Android Applications

To study this matter, it was necessary to develop a set of criteria that fit the goal of this research, which is the definition of Android mutation operators considering the basic guidelines of the background-foreground behaviour in an Android applications. The basis for this criteria is present in [Rib17] since it is a similar research done in the field, whose dataset is extensive an applicable in the context of this study. It was refined having in mind the specific needs of this study.

The final criteria used in this investigation can be observed in table 3.1. The selection of the applications was conducted via the Google Play Store having in mind the criteria previously defined.

An extensive part of the dataset used for this study has also been used in [Rib17]. After taking it into consideration, it was compared against the new criteria defined for this research in order to find possible applications that could also be used for this study.

Table 3.1: Criteria for app selection

Be available in Google Play Store	The application needs to be available in the Google Play Store
Be available in Portugal	The application needs to be available in Portugal to access its information
Be open source	The application's code needs to be available to enable analysis of its code to choose and insert mutants
Have Google Play Store rating >= 3.5	The application's minimum rating is 3.5 to ensure it has some degree of quality
Have Google Play Store downloads >= 10000	The application's minimum downloads is 10000 to ensure it has been used by multiple people
Be Android Native	The applications code must be Android native due to the aim of this work being the definition of Android mutant operators
Use Gradle	The application must use Gradle to simplify the build of the application
Have a GUI	The application must have a Graphical User Interface to be tested by the *iMPAcT tool*
Be in an Western European Language	The application must be in an Western European Language so its UI can be understood

3.1.1 Compiling the Applications

The applications were compiled using Android Studio 3.1 and ran both in an emulator (Nexus 6 with Android 7.0 and Nexus S with Android 7.0) and a real device (Moto E with Android 6.0). All applications that did not build/compile correctly were discarded as this would disable testing the UI of the application in question.

The most common errors building/compiling the applications were:

- Missing google-services.json file
- Gradle sync failed, due to missing files
- Generic build failed error message, due to missing files

In the end, we were left with 50 applications that met all the criteria defined and built/compiled correctly, as observable in table 3.2.

Table 3.2: Application's categories

Category	Number of Apps
Books and Reference	2
Communication	3
Education	2
Finance	1
Library and Demo	3
Lifestyle	1
Maps and Navigation	3
Medical	1
Music and Audio	3
News and Magazines	2
Photography	1
Productivity	6
Puzzle	1
Strategy	1
Tools	14
Travel and Local	4
Video Players and Editors	2
Total	**50**

It was also taken in consideration that the dataset of applications used should be diverse in purpose and use. This is visible in figure 3.2. The category with the most applications is "Tools" since it is the more general one. All the categories represented are according to the Google Play Store.

3.2 Manual Testing

At this stage, it is necessary to understand what is considered correct and incorrect regarding the guidelines (recommendations) for the background-foreground behaviour in Android applications.

An activity is considered to have correct background-foreground behaviour when it is sent to background and brought back to foreground and its overall elements/state are still the same as before. This means that, while other native behaviours of Android applications like screen rotation may lead to a similar but not equal activity screen, when background-foreground behaviour occurs, it is mandatory the screen remains the same as before, except if some loading as been done and this is expected by the app. Figure 3.3 represents an example of correct background-foreground behaviour in an application.

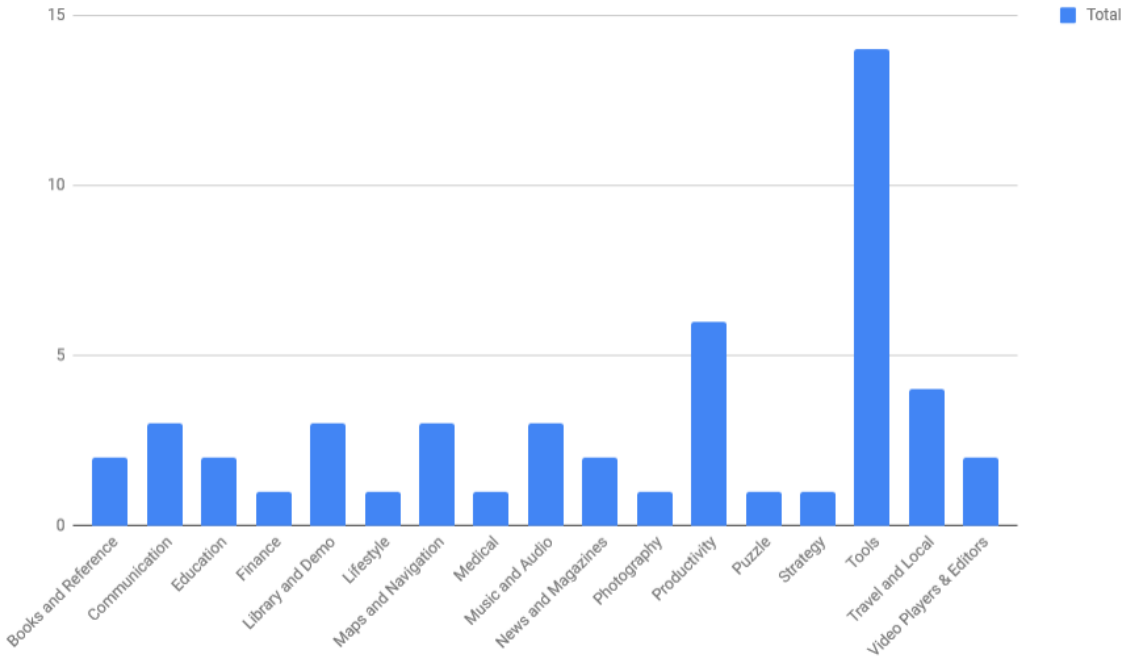
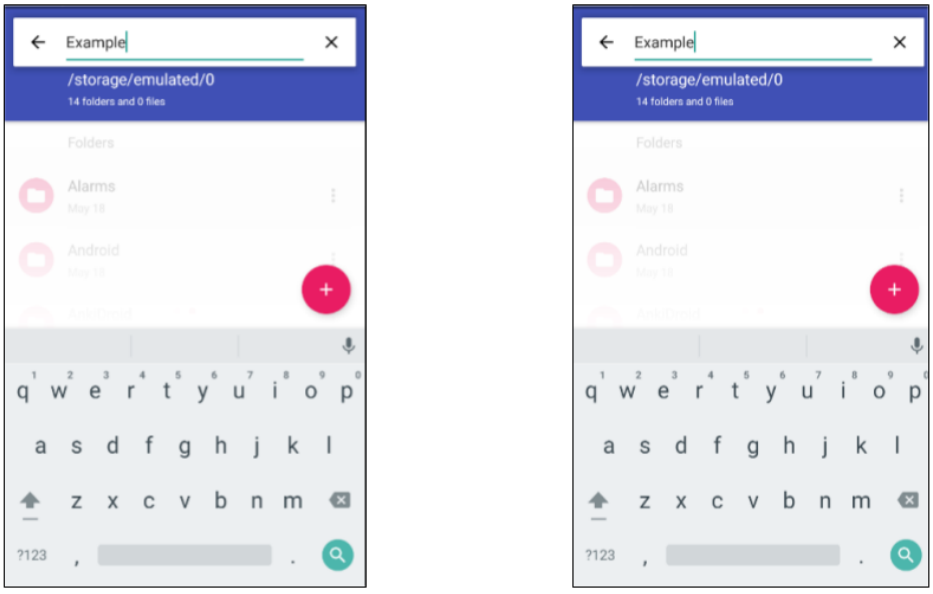


Figure 3.2: Applications distribution according Google Play Stores categories



Before background

After background

Figure 3.3: Example of correct background-foreground behaviour (AmazeFileManager)

As expected, an incorrect background-foreground behaviour happens when the application is sent to background and brought back to foreground and its overall elements/state are no longer the same. Examples of incorrect background/foreground behaviour can be seen in section 3.2.1.

3.2.1 Manual Background-Foreground Behaviour Testing

In the dataset of 50 applications, every single one was manually tested, searching for incorrect background-foreground behaviour. Since going to background and back to foreground is in the nature of all Android applications, it was possible to analyse this behaviour in every application in the dataset. To achieve this, for all applications, every activity was explored using the following process:

1. Open new activity
2. Send activity to background
3. Bring activity to foreground
4. Check if all the widgets that existed before are still present and carry the same information
5. If there is a widget where it is possible to change its state, that action is performed
6. Send activity to background
7. Bring activity to foreground
8. Check if all the widgets that existed before are still present and carry the same information
9. Go back to step 1 until no more activities can be explored

This process was designed according to the Android guidelines present in [Act]. If during this process any error was detected (the application crashed, some action was not possible or some widget lost/gained information after being brought to foreground) the application was considered to have incorrect background-foreground behaviour.

The results obtained by applying this process to all applications in the dataset can be seen in tables A.6 and A.7. The shorten version of these table can be found in table 3.3. 5 defects were found in the background-foreground in 50 applications. This means that 10% of analysed applications have a defect in said behaviour, which is very meaningful to this study and validates its purpose.

Table 3.3: Application's Manual Testing Results against the background pattern

	Number of Apps
Correct	45
Incorrect	5

Figure 3.4 displays the distribution of applications with incorrect background-foreground behaviour detected manually according to their Google Play Store.

The 5 applications with incorrect background-foreground were studied and its faults were classified and distinguished in 3 different types of defects:

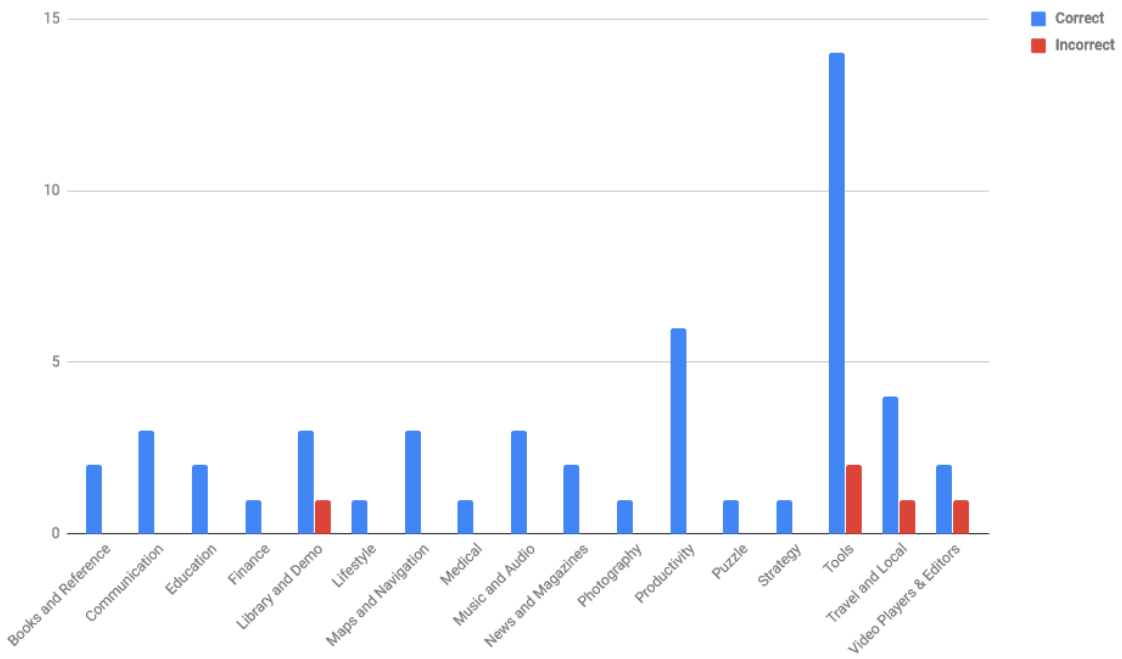


Figure 3.4: Applications correct/incorrect background-foreground behaviour distribution according Google Play Store categories

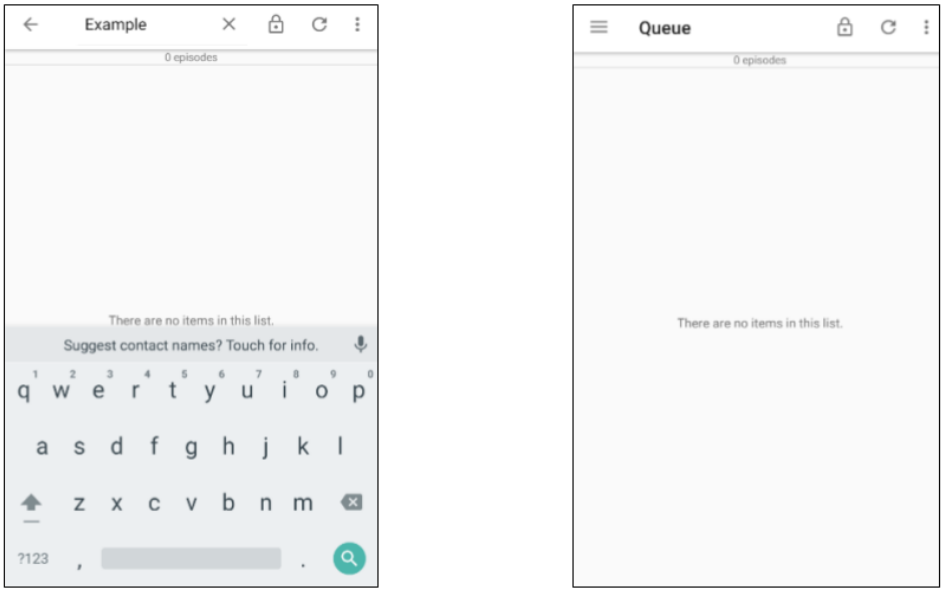
- **State Change** — When an application is in the background and it is brought to foreground, its state has changed, meaning the app is not in the same activity as before or some widget has gained/lost information (Figure 3.5)

- **Widger Appear** — When an application is in the background and it is brought to foreground, a widget has been opened without any input of the user (Figure 3.6)

- **Widget Disappear** — When an application is in the background and it is brought to foreground, a widget has been closed without any input of the user (Figure 3.7)

This division was inspired in [ARPF18] where the authors face a similar problem classifying GUI failures, but for the rotation of the screen. It facilitates the designing of mutation operators, since they can be gathered in these particular groups. By doing so, the way the mutation operators work and how they look in the application screen becomes visible, allowing to understand in general how a certain mutation operator will affect an application.

The 5 incorrect background-foreground behavioured applications were deeply analysed in order to define the mutation operators which will be used for further testing. This information can be found in section 3.4.



Before background

After background

Figure 3.5: Example of state change error (AntennaPod)

3.3 MDroid+ Testing

In order to attest if there really is a need for Android specific mutants, it is paramount to analyse the existing tools used for that purpose. As previously stated, mutation testing in Android is an underdeveloped area, hence the lack of proper tools to aid in its processes.

Since *MuDroid* does not have specific Android mutation operators (only ones applied to general programming derived from Java), we will focus on *MDroid+* and the mutants that are defined in that tool.

To test the effectiveness of *MDroid+*, 15 applications of the dataset where ran against all 39 mutants defined in the tool. These apps were selected in a way so that they would be from several different categories as described in Google Play Store (to ensure they have different scopes) and so that we would have both larger and smaller applications. Only 15 of the 50 applications in our dataset were tested due to time constraints.

The results are divided in 3 different categories according to how the mutant app was evaluated:

- Disruptive — The mutation injected caused the application to crash
- Misfit — The mutation injected did not affect the application's general behaviour and therefore cannot be used in this research
- Valid — The mutation injected created a valid mutant which was detected manually

These results are represented in table 3.4, displaying the amount of mutant apps generated by each operator divided according to the testing result. The observable 7 mutation operators are

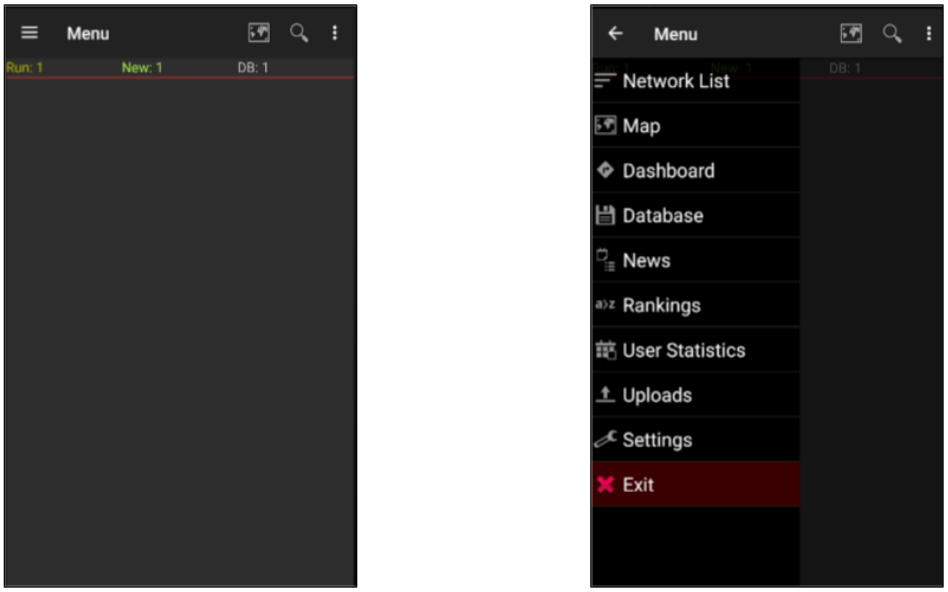


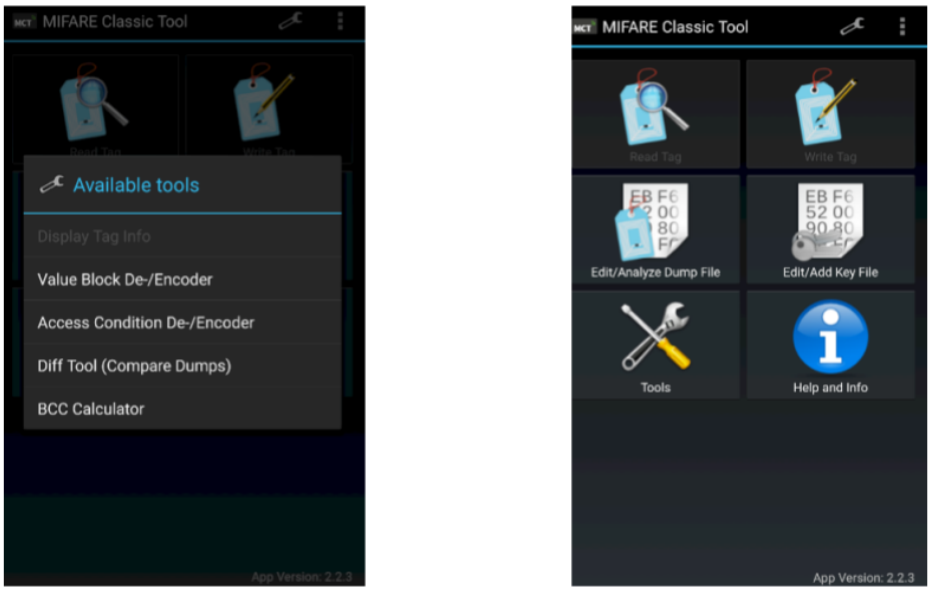
Figure 3.6: Example of widget appear error (WiggleWifi)

the only ones which generated mutant apps by *MDroid+*. The remaining 32 mutation operators did not generate any mutated apps, hence they are not represented in the results table. The tables A.1 and A.2 display the amount of mutants generated per application tested. The tables A.3 and A.4 display the categorization of the mutants generated per application. The app mutants were all tested manually with the exception of the ones generated with *WrongStringResource* mutation operator. Given its large amount of generated mutants, it was too time consuming to test all of them so we chose to test 10% of the mutants generated for each original application. We confidently introduced them all in the *Misfit* category since all tested mutants had similar behaviour and appearance.

Table 3.4: MDroid+ testing results

		Valid	Disruptive	Misfit
Activity Intents	ActivityNotDefined	0	151	0
	InvalidActivityName	0	151	0
	InvalidLabel	0	0	124
	WrongMainActivity	0	0	12
Android Programming	MissingPermissionManifest	0	10	49
	WrongStringResource	0	0	2520
GUI	InvalidColor	0	218	0

After running it against the 15 applications chosen, the *MDroid+* failed to generate the mutants expected. We experimented with each mutant application, but they are based in changing statements that cause the applications crash or replacing variable values that do not affect the overall functionalities of an Android application. Hence, the mutation operators present in the *MDroid+*



Before background

After background

Figure 3.7: Example of widget disappear error (MifareTools)

will not be used for the purpose of this research because they do not alter the functionalities we are targeting.

The *MDroid+* was unable to create any valid mutant apps to this research's goal. None of the mutants affected the specificity of the background-foreground behaviour in Android applications which validates this study even further given the extensive lack of applicable mutation operators that influence the unique behaviours of Android programming.

3.4 Mutation Operators Definition

The analysis of the 5 applications with incorrect background-foreground behaviour and the study of guidelines for good Android programming [Act] allowed to define 4 different mutation operators that affect the background-foreground behaviour of an application.

When the application goes to background, the activity where the user was at that moment calls the method *onPause* and triggers the method *onSaveInstanceState* of the activity. These two methods duties are as follows:

- **onPause()** — This method is called when the activity is in the *Pause* state, meaning that the user is leaving the application (the activity is sent to background). It is responsible handling all necessary actions before the activity goes to background.
- **onSaveInstanceState()** — This method is invoked when the activity may temporarily be destroyed and it is responsible for saving the state of Widgets and the activity as well.

These two methods are key to allow the developer to manage the background-foreground behaviour of an application. Hence, the focal point of the mutation operators defined are these two functions.

The first mutation operator defined consist in clearing the *Bundle outState* being saved in the *onSaveInstanceState* method when the activity goes to background (which contains all the data to be restore when the activity is brought back to foreground). The *outState* represents the *Bundle* carrying the data which will be cleared in the override of *onSaveInstanceState* method in the activity or fragment. If this method is not being overridden in the activity, this mutation operator cannot be injected.

```
outState.clear();
```

Mutant 1 - onSaveInstanceState mutant

The second mutation operator defined consists in disabling the call of the *onSaveInstanceState* method of a certain EditText present in the activity. If there is no EditText in the activity or fragment, this mutation operator cannot be injected.

```
EditText.setSaveEnabled(false);
```

Mutant 2 - EditText mutant

The third mutation operator defined disables the call of the *onSaveInstanceState* method of a certain Spinner. If there is no Spinner in the activity or fragment, this mutation operator cannot be injected.

```
Spinner.setSaveEnabled(false)
```

Mutant 3 - Spinner mutant

The fourth and last mutation operator defined creates an intent to another activity and triggers it in the override of the *onPause* function of the activity (or creates the override of *onPause* if it has not been overridden already), to send the application to another activity when it goes to background. This mutation operator can be injected in all activities which extend any type of *Activity* class (*Activity*, *AppCompatActivity*, etc). It cannot be injected in fragments nor in applications with only one activity which extends any type of *Activity* class.

```
@Override
protected void onPause() {
super.onPause();
android.content.Intent intentMutant = new android.content.Intent(this,About.class);
startActivity(intentMutant);
}
}
```

Mutant 4 - Intent mutant

The second and third mutants (EditText and Spinner) are a segmentation of the first one (onSaveStateInstance), as they disable the call of the *onSaveStateInstance* method (which is the targeted function of the first mutant) for a specific widget. They represent an example of widgets that exist in Android and are influenced by user input. There are other widgets like CheckBox or DialogBox that could also be utilized, but EditText and Spinner were the ones chosen for being more commonly used and understood.

In addition to the fact that the four mutants defined tackle the function responsible for taking care of the background-foreground behaviour in Android applications, programatically they are simple enough to implement as well. By simply searching the activity's file, it is possible to use methods to find and replace statements from the original source code which makes the injection of the mutants very easy and straightforward.

3.4.1 Validation of Mutation Operators

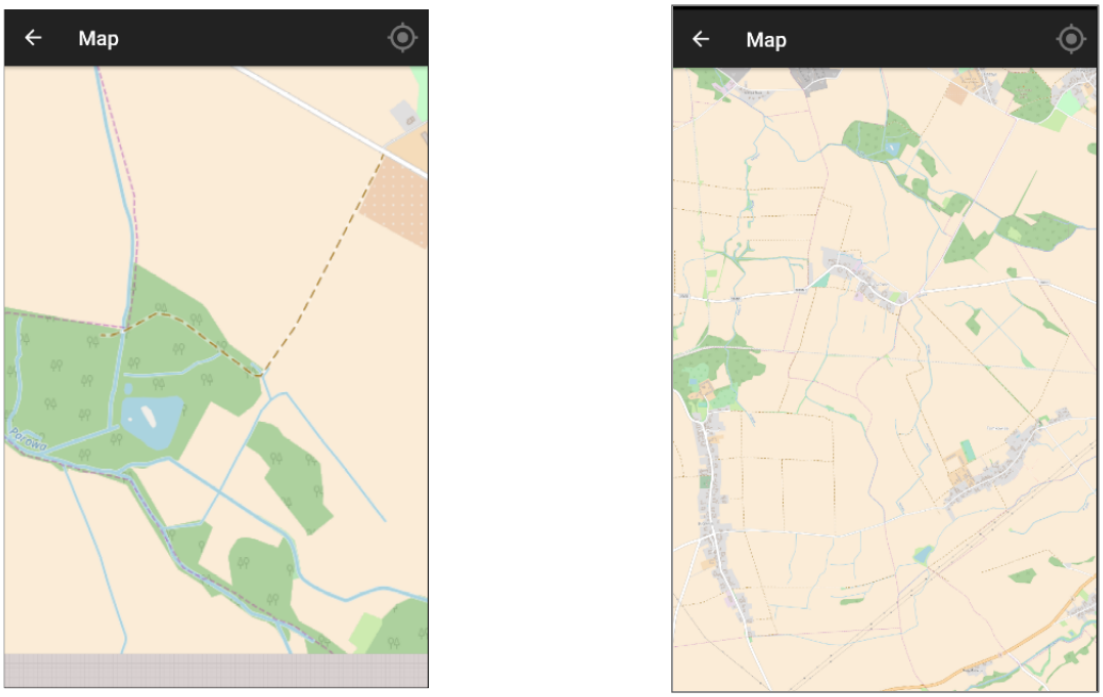
After defining the four mutation operators to affect the background-foreground behaviour of an application, they were validated against the applications present in the dataset of this research which have correct background-foreground behaviour. This validation was achieved by generating several mutant applications which were manually tested to analyse the results of injecting the corresponding mutation operator.

The first mutation operator was validated against applications in the dataset with activities or fragments with the method *onSaveInstanceState* overridden. Figure 3.8 is an example of a mutant application that has been injected with the *OnSaveInstanceState* mutation operator.

The second mutation operator was validated against applications in the dataset with activities or fragments with an EditText instance. Figure 3.9 is an example of a mutant application that has been injected with the *EditText* mutation operator.

The third mutation operator was validated against applications in the dataset with activities or fragments with a Spinner instance. Figure 3.10 is an example of a mutant application that has been injected with the *Spinner* mutation operator.

The forth and last mutation operator was validated against applications in the dataset with at least two activities extending any type of *Activity* class (*Activity*, *AppCompatActivity*, etc). Figure 3.11 is an example of a mutant application that has been injected with the *Intent* mutation operator.



Before background

After background

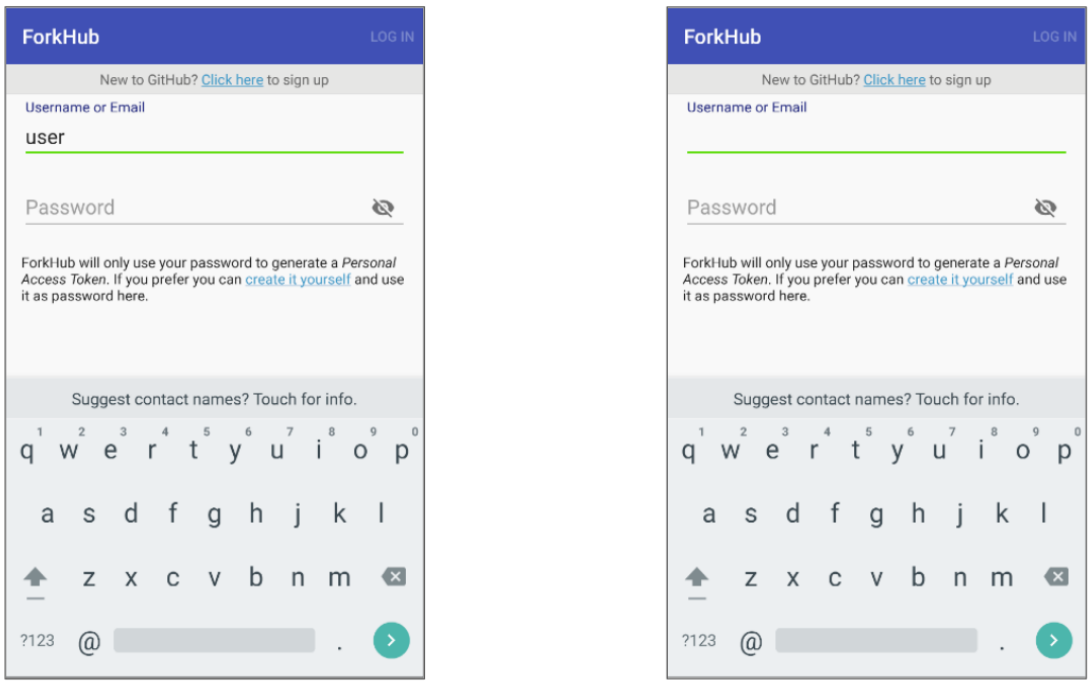
Figure 3.8: Example of a mutated app with the onSaveInstanceState mutation operator injected (OpenBikeSharing)

All the mutation operators defined were successfully validated using the dataset chosen for this research, given all of them created mutant applications with incorrect background-foreground behaviour when injected in the applications of the dataset.

3.4.2 Automation of Mutation Operators

After validating the mutation operators defined, a command line tool was developed using *Node JS* to automatically inject the mutation operators. It works using the following two staged process:

- Exploration
    1. The tool receives the path to the folder of the application where the mutation operators are supposed to be injected
    2. The tool searches the directory for possible files to inject the four mutation operators defined
        — If there are no possible files to inject the mutants, the tool terminates with an explanatory message
        — Otherwise, it proceeds to the following step
- Injection



Before background

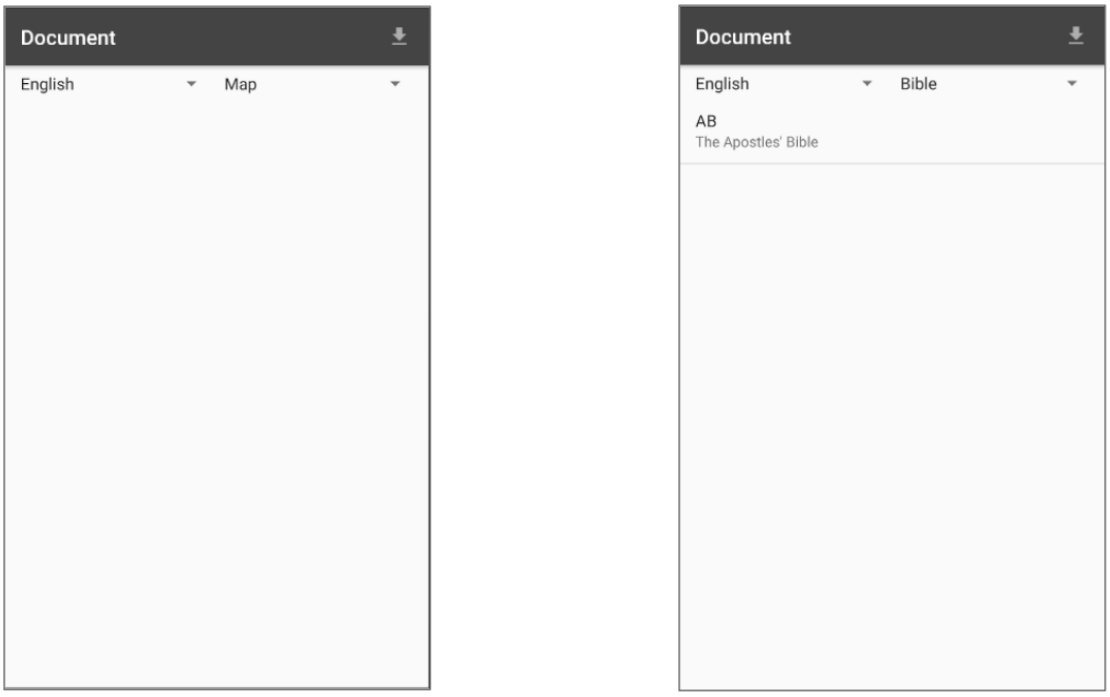
After background

Figure 3.9: Example of a mutated app with the EditText mutation operator injected (Forkhub)

1. For every file where a mutant can be injected, the tool generates a new folder with the content of the original directory and injects the desired mutant there
2. For every mutant injected, the tool prints in the command line the path to the file where the mutant was injected and which type of mutant was injected

Each mutation operator requires different setup and conditions, which means that for each one of the 4 mutants, there are a set of steps that need to be followed. This is how the tool finds where to inject the mutants and how it injects them:

- **Mutant 1 (onSaveInstanceState)** — The tool searches for occurrences of the override of the *onSaveInstanceState* method in activities. When it finds one, it creates a new folder with the content of the original directory and injects the mutant in the new folder, clearing the Bundle outState in the *onSaveInstanceState* method by adding the operator 3.1 to it. The tool outputs the path to the file where the mutant was injected. If it does not find one file to inject the mutant, it outputs that no mutants of the onSaveInstanceState type were injected.
- **Mutant 2 (EditText)** — The tool searches for instances of the EditText widget via a regex expression. When it finds one, it creates a new folder with the content of the original directory and retrieves the instance name in the EditText widget it found. Then, it injects the mutation operator 3.2 which disables the call of the method *onSaveInstanceState* for that widget. The tool outputs the path to the file where the mutant was injected. If it does



Before background

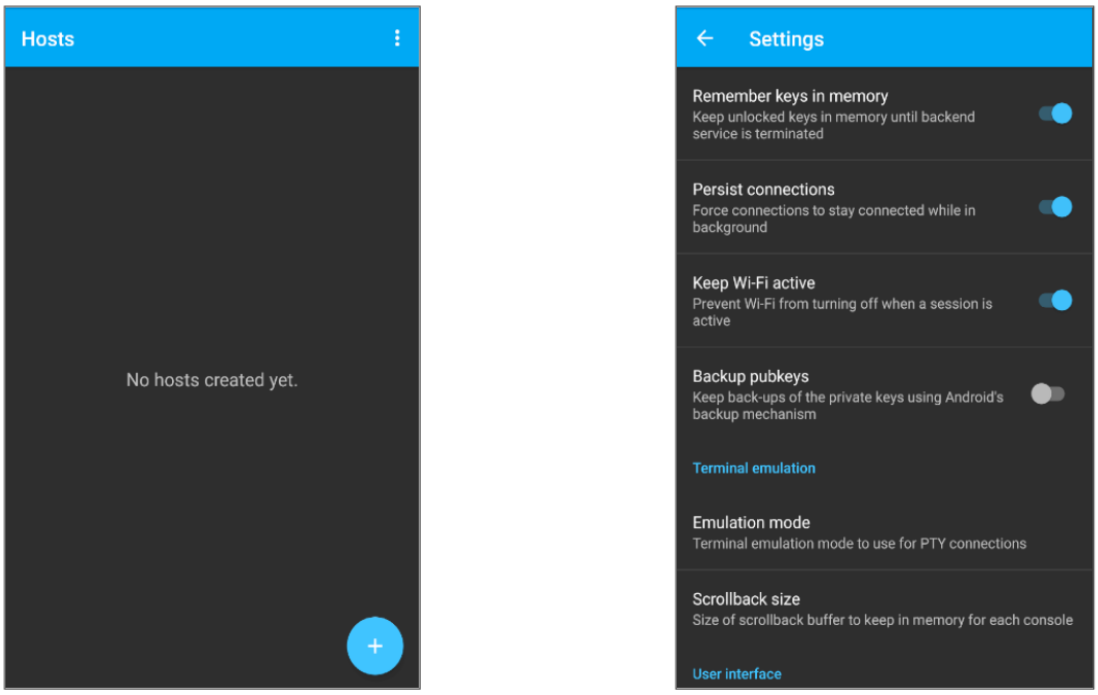
After background

Figure 3.10: Example of a mutated app with the Spinner mutation operator injected (and-bible)

not find one file to inject the mutant, it outputs that no mutants of the EditText type were injected.

- **Mutant 3 (Spinner)** — The tool searches for instances of the Spinner widget via a regex expression. When it finds one, it creates a new folder with the content of the original directory and retrieves the instance name in the Spinner widget it found. Then, it injects the mutation operator 3.3 which disables the call of the method *onSaveInstanceState* for that widget. The tool outputs the path to the file where the mutant was injected. If it does not find one file to inject the mutant, it outputs that no mutants of the Spinner type were injected.

- **Mutant 4 (Intent)** — The tool searches for at least two activities that extend any type of *Activity* class, like *Activity* or *AppCompatActivity*. If there are at at least two activities that check these conditions, for each one of them it creates a new folder with the content of the original directory. In the file of the first activity it searches for the override of the *onPause* method (if there is none, the tool creates it) and injects the mutant triggering an intent to the second activity inside this function with operator 3.4, which causes the application to jump to a new activity upon going to background. The tool outputs the path to the file where the mutant was injected. If it does not find one file to inject the mutant, it outputs that no mutants of the Intent type were injected.



Before background

After background

Figure 3.11: Example of a mutated app with the Intent mutation operator injected (connectBot)

Note that the tool does not generate the mutant APK needed to test the application. The tool outputs folders with the injection of each mutant, containing all files/folders inside the directory given as input. In order to generate the mutant APK, it is necessary to first substitute the original directory given with the mutant folder.

The search for each type of mutant is done separately, beginning in the first one until the forth in order. If a certain type of mutant could not be injected the output of the tool goes accordingly to listing 3.5, for each mutant type unable to be injected.

```
mutant_type could not be injected
```

Listing 3.5: Tool's output when not possible to inject mutant of certain type

If the there are locations to inject a certain type of mutant, the output of the tool goes accordingly to listing 3.6, for each mutant type that is injected.

```
mutant_type
path_to_file_where_mutant_is_injected
```

Listing 3.6: Tool's output when possible to inject mutant of certain type

Also, in case the user messes up the original directory of the application where the mutants are being injected, the tool generates a copy of the original folder to act as backup, that will be kept unaltered.

The tool is rather quick in its search and injection of mutants, ranging from 5 seconds (in case the directory to search is small and no mutants can be injected) up to 60 seconds per execution (in case the directory to search is big and there are a lot of locations where mutants can be injected).

The tool's general behaviour and output was inspired in the *MDroid+* tool, which also receives a path to a directory and injects the selected mutants, creating separate folders for the injection of each one. This aids in the process of building/compiling the mutant apps due to the easy design of separation between each mutant injected.

The tool developed is prepared for the automation of more mutation operators, given its well defined structure. Most mutants would simply require the replication of the mutation operators already defined with a few adjustments.

3.4.2.1 Liabilities

Regarding the first and fourth mutants (onSaveInstanceState and Intent), the tool is well implemented and can find every location to inject them. The same cannot be said about the second and third mutants (EditText and Spinner), since the implementation of the tool can only find the first location to inject a certain type of mutant per file. Hence, if an activity has more that one instance of EditText or Spinner, the tool will only be able to inject a mutant in their first instance.

Another setback is besides being implemented using NodeJS, for time constraints, the development of this tool was done without using one of the most valuable assets of NodeJS which are the asynchronous callbacks. If implemented using them, the tool's performance would be even better, given that multiple mutants could be injected at the same time, due to not having to wait for finishing the injection of a mutant to start the finding and injection of the next one. This would make this process much faster.

Also, the tool does not allow to choose which type of mutation operators will be injected, given that this selection is still hardcoded. This feature would bring value to the tool, because the user may want a specific mutation operator to test a certain test suite and this way it would be possible to achieve just that. This would make the process of running the tool against an application faster if we only wanted to inject a certain type of mutation operator.

As a final remark, the tool does not create a log file so that the user could analyse its results after closing the command line. This feature would also be important to keep track of the tool's behaviour for each application.

In spite of having a few issues, the tool's behaviour is satisfying and it stands as a benefit to the automated mutation injection scene. The possibility to generate mutants that affect a specific Android behaviour is a great asset which is quick and effortless.

3.5 Conclusion

In this chapter, we analysed 50 different applications and how their background-foreground action behaved. 5 of those 50 apps were deemed to have faulty background-foreground behaviour, having those faults been divided in 3 categories, according to their nature.

Also 4 mutation operators that affect the background-foreground behaviour in Android applications were defined taking into consideration the defects in the background-foreground behaviour among the applications in the dataset and the guidelines for correct Android programming. Accordingly, these mutation operators can be used to assess the effectiveness of test suites in Android applications and other testing tools alike.

All the 4 mutation operators were successfully automated in order to quickly generate mutant apps capable of validating the test suites defined in Android applications.

For this purpose, and recurring to the automated mutation testing tool developed using these mutation operators to generate mutant apps of our original dataset, the iMPAcT tool's background pattern will be evaluated regarding its ability to detect background-foreground induced incorrect behaviours, which will happen in chapter 4.

Chapter 4

Case Study

In this chapter it is presented the results of this research. For each application, we generated the possible mutant apps using the automated mutation tool presented in section 3.4.2. The generated mutants were ran against the background pattern defined in the *iMPAcT tool* to attest its effectiveness and performance. An extra test was done, which consisted in running the original dataset against the background pattern defined in the *iMPAcT tool* to further assess its efficiency.

4.1 Automated Mutation Injection

All applications were ran through the automated mutation injection tool developed in section 3.4.2 to observe how many mutants were possible to inject in each application. The results presented in table 4.1 are an overview on the tool's performance. A full table can be found in A.5 with information per application displayed.

Table 4.1: Number of Mutant Apps Generated for each Mutation Operator

Mutation Operators	**Mutants Generated**
onSaveInstanceState	112
EditText	36
Spinner	19
Intent	466

It is visible that the mutation operators *onSaveInstanceState* and *Intent* generate a much more significant number of mutants for the 50 applications in the dataset than the mutation operators *EditText* and *Spinner*. This happens because:

- The *onSaveInstanceState* mutant is applicable to all activities and fragments in an application that are overriding the *onSaveInstanceState* method, which tends to happen because that method is what is used by programmers to manipulate the saving of states and properties of an activity and its widgets before it goes to background.

- The *Intent* mutant is applicable to all activities in an application that extend any type of *Activity* class, if there are at least two activities with that property, which happens in most applications. This mutation operator is injected in each application the number of activities that extend any type of *Activity* class. In big and complex applications with loads of different screens (with each screen representing the display of a different activity), this translates in a high number of locations to inject the *Intent* operator and consequently, a high number of mutants injected.

- The *EditText* and *Spinner* mutants are only applicable to activities and fragments with EditText/Spinner instances, which is not so common amongst the applications of our dataset. Since they are a particular case of the *onSaveInstanceState* mutant, they are injected in a much smaller scale.

Regardless of the low mutant apps generated, the *EditText* and *Spinner* mutants are also considered important due to their specificity. These two mutation operators tackle the distinct case of widgets which the user can interact with and are also an asset to the mutation operators developed in this research.

In table 4.2 it is visible the number of applications where it was possible to inject each type of mutation operators. The *Intent* mutation operator was by far the one which generate the most mutant apps per application for the original dataset. This is due to its easier preconditions that need to be met by the application, as described in section 3.4. The other mutation operators follow the same idea behind its capability of generating mutant apps, given the preconditions stated earlier.

As a further validation of the mutation operators defined only in 3 applications (uCrop, Wifi-Automatic and PhotoAffix) no mutant application was generated, which means that we generated mutant apps for 47 apps, out of 50. This is a very good indicator in favour of the mutation operators defined.

Table 4.2: Number of applications where was possible to inject each type of mutation operator

Mutation Operators	**N^{o} of Applications**
onSaveInstanceState	22
EditText	15
Spinner	12
Intent	47

All mutation operators defined and automated successfully generated an acceptable number of mutant apps from the original 50 applications in the dataset. Only in 3 applications was not possible to inject mutants.

Given the large amount of mutants generated, in a total of 633 mutant applications with the injection of 4 mutation operators, it is possible to state that the tool's mutation operators inject

mutants spread throughout the code of the apps. This will force the test suites to be more specific which will help to improve the quality of testing in Android applications.

4.2 iMPAcT Tool Testing

In order to assess the capability to detect incorrect background-foreground behaviour of the testing tool chosen for the purpose of this research (*iMPAcT tool*), some experiments were done against the background pattern, which is defined in the *tool* itself.

To detect failures in Android applications, as a testing tool that uses pattern matching, reverse engineering and screenshot techniques, the *iMPAcT tool* explores the AUT searching for incorrect behaviours. So it can find and kill the mutants injected in each application, the *iMPAcT tool* must acknowledge the incorrect behaviour introduced (depending on which pattern is being tested) by the mutation operator. This is done by successfully detecting the faulty behaviour in the screenshots taken during the applications exploration.

At first, the mutant applications generated by the automated mutation tool developed during this study described in section3.4 were ran against the background pattern. This was done to test the *iMPAcT tool* against applications which were deliberately incorrect in their background-foreground behaviour to study its effectiveness in detecting faulty applications.

At last, the original dataset of applications was ran against the background pattern present in the *iMPAcT tool* to observe if it would find the same incorrect behaviours as we did manually or if it could find more incorrect behaviours other than those. This was done to further analyse the *iMPAcT tool*'s testing strategies as well as our dataset.

The exploration technique used in the *iMPAcT tool* to investigate each application of the dataset was PRIORITY_TO_NOT_EXECUTED. This technique was the one chosen to maximize the exploration of the AUT [MP16]. The applications were ran in a real device (Moto E with Android 6.0). The pattern tested with the *iMPAcT tool* was the background pattern. The *iMPAcT tool* ran for about 5-20 minutes per mutant application tested.

4.2.1 Background Pattern Testing

In order to analyse the results of running the mutant applications against the background pattern defined in the *iMPAcT tool*, it is necessary to understand what is being tested and how the *tool* detects failures.

When a user exits an application via the home button, the app goes to background and the screen returns to the home menu. At this time, the application should save its current state in case the user goes back to the same application since pressing the home button does not fully exit it. When the application goes to the foreground again, it is expected that its state remains the same as it was when the application was first sent to background, meaning that all the same elements should exist, be in the same place and carry the same information.

To achieve this the *iMPAcT tool* performs the following steps:

- Checks if the AUT goes to background when pressing the home button. By doing this the process should not be killed and the application should not crash.
- When the application is brought from background to foreground, checks if its state is the same previous to being sent to background.

The background-foreground behaviour exists in Android applications by nature. Hence, there is no need to define the UI pattern formally since the only condition to perform this test it that the AUT must be opened.

Therefore it is only necessary to formally define the Test Pattern, as stated in [Fer17]:

```
Goal: "App goes to background and keeps the same state from before background event
    "
V: {}
A: ["observation", app goes to background, "observation", send app to foreground, "
    observation"]
C: {"Verify the app state is equal to the previous one"} P: {"AUT is open && TP not
     applied to current activity"}
```

If an application fails the test strategy, it will be considered to have the background pattern incorrectly implemented.

Each mutated app was ran against the background pattern present in the *iMPAcT tool* to analyse if the incorrect behaviours introduced were caught by the *tool*.

The results were divided in 4 categories according the way the mutant app behaved and if the faults injected were detected or not:

- **Not Spotted** — The failure was not detected neither manually nor via the *iMPAcT tool*
- **Not Detected** — The failure was detected manually but not via the *iMPAcT tool*
- **Detected** — The failure was detected both manually and via the *iMPAcT tool*
- **Incorrect** — The injection of the mutant caused the application to crash

The results are represented in table 4.3. As a note, due to time constraints, only about 30% of the Intent type mutant apps were tested against the background pattern. We confidently state all of them as detected because:

- The mutants generated with the *Intent* operator were very similar in appearance and behaviour
- We tested at least 3 mutants apps generated with the *Intent* operator of each original application in the dataset
- All the tested mutant apps generated with the *Intent* operator were detected by the *iMPAcT tool*

- The difference between the screen before the activity went to background and the one after it came back to foreground in mutant apps generated with the *Intent* operator was too extreme for the *iMPAcT tool* to miss it

It is also important to state that there were no applications where the *iMPAcT tool* detected incorrect behaviour that could not also be found manually. This is very important because it means that if the *iMPAcT tool* detects faulty behaviour, this faulty behaviour really exists in the application being tested.

Table 4.3: Mutated Application's *iMPAct tool* Testing Results

	onSaveInstanceState	*EditText*	*Spinner*	*Intent*
Not Spotted	28	12	3	0
Not Detected	6	4	2	0
Detected	78	20	14	457
Incorrect	0	0	0	9
Total	112	36	19	466

As observable, the *iMPAcT tool* found the great majority of the mutants injected (569 in total).

There are 43 "Not Spotted" mutants that were not detected manually nor by the iMPAcT tool, meaning that these mutants are equivalent. These mutants were generated by the mutation operators *onSaveInstance*, *EditText* and *Spinner*. The likely reason behind this result is related to the different ways a developer can preserve and restore an activity's UI state. This can be achieved by implementing a *view model*, *save the instance state* or use *persistent storage*. So, in case the app uses one of the other two methods to save UI state other than via the *onSaveInstanceState* method, the mutation will have no effect, meaning that it will generate an equivalent mutant of the original app. Also, since the *onSaveInstanceState* method is triggered when the activity is briefly destroyed, in case this does not happen, the mutant will be equivalent as well. Hence there is no test case that is able to distinguish and detect these mutant apps.

There are 9 "Incorrect" mutant apps that crashed while being tested. They were generated by injecting the *Intent* mutation operator. This mutation operator is the one which causes more disturbance in the background-foreground behaviour by sending the app to a different activity. If before entering an activity, the application needs to have some conditions met and this does not happen, if the error handling is not correctly implemented, the application will likely crash. For example, if the application tries to open an activity which needs to have the user logged in and this condition is not met, the application will crash without the proper error handling. This might be the reason for the number of "Incorrect" mutant apps, which happened all in the same application (Primitive FTPd).

There are 569 "Detected" mutant apps whose incorrect background-foreground behaviour was successfully detected by the *iMPAcT tool*. These applications were tested both manually and against the background pattern defined in the *iMPAcT tool*. They were generated by all defined mutation operators (*onSaveInstanceState*, *EditText*, *Spinner* and *Intent*). This further validates the

mutation operators defined in section 3.4 because every operator was capable of generating valid mutant apps which could be tested against a UI pattern testing tool.

There are 12 "Not Detected" mutant apps whose incorrect background-foreground behaviour was not detected by the *iMPAcT tool*, but its faulty behaviour was detected via manual testing. This means that, while the *iMPAcT tool* is detecting most of the incorrect behavioured applications, it cannot detect all of them. The reasons behind this will be further explored in section 4.2.2.

4.2.2 *iMPAcT tool*'s evaluation

Observing the results in table 4.3, it is possible to extract some valuable information about how the *iMPAcT tool* is detecting and what are its potential problems regarding its test strategies.

The mutant applications whose incorrect behaviour was the change of elements on the screen, generated by the *Intent* mutation operator, were all successfully detected via the *iMPAcT tool*.

The mutant applications that the *iMPAcT tool* did not correctly identified as incorrect behavioured are the ones centered around the loss of information by the activity and deal with user input, generated by the mutation operators *onSaveInstanceState*, *EditText* and *Spinner*.

This means that while the *iMPAcT tool* is correctly identifying applications where the background-foreground behaviour is changing elements on the screen (either the state or the position), it is not identifying all widgets that lose information upon going to background and returning to foreground.

Given the results presented in table 4.3, it is possible to analyse the overall efficiency of the *iMPAcT tool* in detecting incorrect background-foreground behaviour injected by each mutation operator (the "Incorrect" results are not considered because these mutant applications crashed while being tested), by calculating the Mutation Score according to the formula in 2.4.1:

Mutation Score (MS) = Mutants killed / (Total Mutants - Equivalent Mutants)

The results of calculating the Mutation Score per mutation operator and overall are as follows:

- *onSaveInstanceState* — Mutation Score = 78 / (112 - 28) = 0.929
- *EditText* — Mutation Score = 20 / (36 - 12) = 0.838
- *Spinner* — Mutation Score = 14 / (19 - 3) = 0.875
- *Intent* — Mutation Score = 477 / (477 - 0) = 1
- Considering all the mutants — Mutation Score = 569 / (624 - 43) = 0.979

The Mutation Score results are very good. Every Mutation Score is above 0.8, with the Mutation Score for the *Intent* operator being 1 (perfect). Although the overall Mutation Score for all the mutants is 0.979, this number is misleading because of the high number of mutant apps injected with the *Intent* operator.

To further analyse the *iMPAcT tool*'s competence in finding background-foreground behaviour, a second test was done. Given the entire 50 original applications from the dataset, they were each

ran against the background pattern present in the *tool* and compared with the results from the manual testing of the same.

Table 4.4: Comparison of Manual Testing against *iMPAcT tool* testing of the original dataset

	Manual Testing	iMPAcT tool Testing
Correct	45	50
Incorrect	5	0

As observed in table 4.4, all applications were considered to have the background pattern as defined in the *iMPAcT tool* correctly implemented. But as it is possible to analyse, some errors regarding the background-foreground behaviour were found during the manual testing phase.

From this we can examine that the *iMPAcT tool* is not recognizing incorrect background-foreground behaviour 5 times, corresponding is 10% of our dataset. This is a considerable value that cannot be overlooked. The *tool* failed to identify any of the background-foreground incorrect behaviours which were found manually in the original dataset.

The inability to detect these incorrect background-foreground behaviours can be related to different factors analysed in the *tool*'s log:

- The test strategy defined for the background pattern in the *iMPAcT tool* might not be finding all problems related to the background-foreground behaviour (one documented problem is that the *iMPAcT tool* does not detect loss of information in the search widget [Rib17])
- The *iMPAcT tool* might not be exploring the activity where errors were manually found since the exploration technique used does not ensure the testing of every activity in the application
- Some of the incorrect behaviours found in the original applications are abnormal, which makes them difficult to find (like the application Lottie which upon going to background and returning to foreground disables the typing on text input widgets)

These problems need to be analysed in order to make the *iMPAcT tool* capable of detecting every problem concerning the background pattern. The fact that the *tool* does not detect loss of information in the search widget is problematic since most applications have search widgets to allow the user to navigate through the data contained in the app.

Other than this, the overall efficiency of the *iMPAcT tool* in detecting incorrect background-foreground behaviours is very satisfactory. The *tool* detected most of the incorrect behavioured applications, hence we can affirm that its use is beneficial and favourable to testing of Android applications and software quality in general.

4.3 Conclusion

In this chapter, we analysed the automation of the mutation operators defined in section 3.4 and the results of running the background pattern defined in the *iMPAcT tool* against the generated mutant applications.

The automation of the mutation operators proved to be successful as all mutants allowed the generation of several valid mutant apps to be tested. The *Intent* mutation operator was the one which generated the most amount of mutants given that most applications meet its preconditions. The *EditText* and *Spinner* mutation operators were the ones which generated the least amount of mutant apps, because given their specificity in user input widgets, they do not exist so often in applications. The *onSaveInstanceState* mutation operator stands as the generalization of these two. The tool developed only generated a few equivalent mutants, which demonstrates the effectiveness of the mutation operators defined. Also, only 9 of the 633 mutant apps generated crashed when tested which demonstrates that the mutants defined are not disruptive to Android applications upon injection.

Testing the mutant apps against the background pattern present in the *iMPAcT tool* was a labour intensive and time consuming task since the *tool* can take up to 20 minutes to run against each mutant application.

The results of testing the mutant applications are rather positive. The *iMPAcT tool* was able to detect most of the mutant applications, especially when the incorrect behaviour was related to changing the elements on the screen. The cases where the *iMPAcT tool* failed to detect the incorrect behaviour in the background-foreground (when widgets on the screen lost information) were likely due to imprecision in its test strategy for the background pattern.

Besides this, the *iMPAcT tool*'s capability to detect the mutation operators injected was great and it stands as a valuable asset to test UI patterns present in Android applications.

Chapter 5

Conclusions and Future Work

In this research, we defined four different mutation operators capable of affecting the background-foreground behaviour of Android applications, which exists in these apps by nature. They were defined taking into consideration the guidelines and good practices of Android programming.

These mutation operators were injected in several Android applications with correct background-foreground behaviour in order to be validated. This was very successful since all mutation operators introduced faulty background-foreground behaviour in the applications in which they were injected.

Afterwards, we proceeded with the automation of said mutation operators in a Node JS tool and ran the tool against the dataset of Android applications defined at the beginning. The tool searches for locations where the mutation operators can be injected and generates a new folder containing the input directory with the mutant injected. Due to time constraints it was not possible to optimize the tools performance (using asynchronous callbacks). This would speed up the run time of the tool for each application because it would not have to wait for the injection of one mutation to inject the next. In spite of this, the performance of the tool is satisfying since its run time is at maximum about 1 minute.

All mutation operators generated a significant amount of mutant applications with incorrect background-foreground behaviour. Only a few mutant applications were equivalent or disruptive upon testing. This validates the mutation operators defined even further, the tool developed itself and its purpose.

The next step was to run the mutant applications generated against the background pattern defined in the *iMPAcT tool*. The exploration mode chosen was "PRIORITY_TO_NOT_EXECUTED" to ensure the target application was examined as much as possible. The time it took for the *tool* to test each mutant ranged from 5 to 20 minutes.

Analysing the results obtained by testing each mutant against the *iMPAcT tool*, we verified that in some cases it cannot detect loss of information in user input widgets. Also, it is likely

that the *tool* cannot reach every activity/screen in the application which means that if an incorrect behaviour exists in said activity, the *tool* will not detect it.

In spite of this, the *iMPAcT tool*'s capability of detecting incorrect background-foreground behaviour is considerable since it detected most of the mutant apps generated by the mutation operators developed in this study.

5.1 Goal Satisfaction

The main goal of this research was to define a set of mutation operators that affected the background-foreground behaviour in Android applications and also automate them. This was very successful since by following the guidelines and good practices of Android programming we were able to define mutation operators that caused the application in which they were injected to have incorrect background-foreground behaviour. The automation stage was also accomplished by developing a Node JS tool which injected automatically the mutation operators in the input directory.

These mutation operators enabled the subsequent assessment of the quality of the *iMPAcT tool*s background pattern testing strategy. By running the mutant apps generated against the *tool* we found out two problems, one regarding the *tool*s overall functioning and other in the test strategy defined for the background pattern. The first is that the *iMPAcT tool* able to reach all the activities/screens in an application, which may lead to undetected incorrect behaviours. The later is that for some input widgets (namely *EditText*, *Spinner* and as documented in [Rib17], the *Search* widgets), the *iMPAcT tool* cannot detect the loss of information when the application goes to background and comes back to foreground.

5.2 Future Work

As future work it would be interesting to define more mutation operators related to input widgets to analyse if there is one that the *iMPAcT tool* can always detect the loss of information. These mutation operators should also be implemented in the tool developed during this study. Another improvement to make in the tool developed would be to refactor it and add asynchronous callbacks to enhance its performance. Also, it would be important to give the user the ability to choose which mutation operators to inject in the application. In a later stage, the development of a user friendly GUI would also be significant.

As far the *iMPAcT tool* goes, it would be crucial to analyse the problems stated in this study regarding its functioning and background test strategy because as it is, some incorrect background-foreground behaviours go undetected.

It would also be important to extend the dataset of tested Android applications to have a better idea on how the size of the applications influence both the runtime of the tool developed and the *iMPAcT tool* and their ability to achieve its purpose, namely injecting mutants in desired locations and detecting incorrect behaviours in Android applications.

References

[Act] The Activity Lifecycle | Android Developers. https://developer.android.com/guide/components/activities/activity-lifecycle.html. [Online; accessed 22-01-2018].

[AFT11] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A GUI crawling-based technique for android mobile application testing. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, pages 252–261, 2011.

[Anda] Android vitals | Android Developers. https://developer.android.com/topic/performance/vitals/index.html. [Online; accessed 22-01-2018].

[Andb] App Components | Android Developers. https://developer.android.com/guide/components/index.html. [Online; accessed 04-02-2018].

[Andc] Broadcasts | Android Developers. https://developer.android.com/guide/components/broadcasts.html. [Online; accessed 04-02-2018].

[Andd] Content Providers | Android Developers. https://developer.android.com/guide/topics/providers/content-providers.html. [Online; accessed 04-02-2018].

[Ande] Services | Android Developers. https://developer.android.com/guide/components/services.html. [Online; accessed 04-02-2018].

[Andf] Test Your App | Android Studio. https://developer.android.com/studio/test/index.html. [Online; accessed 22-01-2018].

[App] Appium. Appium: Mobile App Automation Made Awesome. http://appium.io/. [Online; accessed 22-01-2018].

[ARPF18] Domenico Amalfitano, Vincenzo Riccio, Ana C.R. Paiva, and Anna Rita Fasolino. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Software Testing Verification and Reliability*, 28(1):1–27, 2018.

[CMPF12] Inês Coimbra Morgado, Ana Paiva, and João Faria. Dynamic Reverse Engineering of Graphical User Interfaces. *International Journal on Advances in Software*, 5(3):224–236, 2012.

[CPN14] Pedro Costa, Ana C.R. Paiva, and Miguel Nabuco. Pattern Based GUI Testing for Mobile Applications. *2014 9th International Conference on the Quality of Information and Communications Technology*, pages 66–74, 2014.

[DMAO] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. MDroid+. http://android-mutation.com/. [Online; accessed 03-02-2018].

[DMAO15] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. Towards Mutation Analysis of Android Apps. *ICSTW 2015, 2015 IEEE 8th International Conference on Software Testing, Verification and Validation Workshops*, pages 1–10, 2015.

[DOA14] Marcio Eduardo Delamaro, Jeff Offutt, and Paul Ammann. Designing deletion mutation operators. *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, pages 11–20, 2014.

[DP17] Fernando Dias and Ana C.R. Paiva. Pattern-Based Usability Testing. *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2017*, pages 366–371, 2017.

[Eng] Engadget. Google's Play Store will boost rankings of high quality apps. https://www.engadget.com/2017/08/03/google-play-store-boost-rankings-high-quality-apps/. [Online; accessed 22-01-2018].

[Exe] UI/Application Exerciser Monkey | Android Studio. https://developer.android.com/studio/test/monkey.html. [Online; accessed 22-01-2018].

[Exp] Espresso | Android Developers. https://developer.android.com/training/testing/espresso/index.html. [Online; accessed 22-01-2018].

[Fer17] Ana Rita Silva Ferreira. Android Testing. Master's thesis, 2017.

[JH11] Yue Jia and M Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[JMAP18] Paulo J. M. Araújo and Ana Paiva. Pattern based web security testing. *Proceedings - 6Th International Conference on Model-Driven Engineering and Software Development, Modelsward, Funchal, Madeira - Portugal, January, 2018*, pages 472–479, 01 2018.

[Kot] Kotlin and Android | Android Developers. https://developer.android.com/kotlin/index.html. [Online; accessed 03-02-2018].

[Lif] Lifewire. What Is the Android Operating System? https://www.lifewire.com/what-is-google-android-1616887. [Online; accessed 22-01-2018].

[LVBT+17] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Enabling Mutation Testing for Android Apps. 2017.

[MdFE12] Henry Muccini, Antonio di Francesco, and Patrizio Esposito. Software testing of mobile applications: Challenges and future research directions. *7th International Workshop on Automation of Software Test (AST 2012)*, pages 29–35, 2012.

[Moc] Mockito. Mockito framework site. http://site.mockito.org/. [Online; accessed 22-01-2018].

[Mon] monkeyrunner | Android Studio. https://developer.android.com/studio/test/monkeyrunner/index.html. [Online; accessed 22-01-2018].

[Mor17] Inês Coimbra Morgado. *Automated Pattern-Based Testing of Mobile Applications*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, 2017.

[MP14] R.M.L.M. Moreira and A C R Paiva. A GUI modeling DSL for pattern-based GUI testing PARADIGM. *ENASE 2014 - Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 126–135, 2014.

[MP15] Ines Coimbra Morgado and Ana C R Paiva. Test Patterns for Android Mobile Applications. *Proceedings of the 20th European Conference on Pattern Languages of Programs*, pages 1–7, 2015.

[MP16] Inês Coimbra Morgado and Ana C.R. Paiva. The iMPAcT tool: Testing UI patterns on mobile applications. *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 876–881, 2016.

[MPM13] Rodrigo M L M Moreira, Ana C R Paiva, and Atif Memon. A pattern-based approach for GUI modeling and testing. *2013 IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013*, pages 288–297, 2013.

[MPNM17] Rodrigo M. L. M. Moreira, Ana Cristina Paiva, Miguel Nabuco, and Atif Memon. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability*, 27(3):e1629, 2017.

[NP14] Miguel Nabuco and Ana C. R. Paiva. Model-based test case generation for web applications. In Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, Carmelo Torre, Jorge Gustavo Rocha, Maria Irene Falcão, David Taniar, Bernady O. Apduhan, and Osvaldo Gervasi, editors, *Computational Science and Its Applications – ICCSA 2014*, pages 248–262, Cham, 2014. Springer International Publishing.

[Off89] A. Jefferson Offutt. The Coupling Effect: Fact or Fiction. *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, 1989.

[Off92] A J Offutt. Investigation of the software testing coupling effect. *ACM Transaction on Software Engineering Methodology*, 1(1):3–18, 1992.

[PMVS14] C Mano Prathibhan, A Maliani, N Venkatesh, and K Sundarakantham. An automated testing framework for testing android mobile applications in the cloud. *IEEE International Conference on Advanced Communication Control and Computing Teclmologies (ICACCCT)*, (978):1216–1219, 2014.

[Rib17] Liliana Filipa Lobo Ribeiro. Injeção de Defeitos em Aplicações Android. Master's thesis, 2017.

[Rob] Robotium. Robotium. https://github.com/RobotiumTech/robotium. [Online; accessed 22-01-2018].

[Sau] SauceLabs. Mobile Device Emulator and Simulator vs Real Device | Sauce Labs. https://saucelabs.com/blog/mobile-device-emulator-and-simulator-vs-real-device. [Online; accessed 22-01-2018].

[Sof] Beginner's Guide to Mobile Application Testing — Software Testing Help. http://www.softwaretestinghelp.com/beginners-guide-to-mobile-application-testing/. [Online; accessed 22-01-2018].

[Sta] Number of smartphone users worldwide 2014-2020 | Statista. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. [Online; accessed 23-01-2018].

[Tec] Broadcasts | Android Developers. https://developer.android.com/studio/test/monkeyrunner/index.html. [Online; accessed 04-02-2018].

[UIA] UI Automator | Android Developers. https://developer.android.com/training/testing/ui-automator.html. [Online; accessed 22-01-2018].

[Ver] Google announces over 2 billion monthly active devices on Android - The Verge. https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users. [Online; accessed 22-01-2018].

[WA15] Yong Wang and Yazan Alshboul. Mobile security testing approaches and challenges. *2015 First Conference on Mobile and Secure Services (MOBISECSERV)*, pages 1–5, 2015.

[Wei] Yuan Wei. MuDroid: Mutation Testing for Android Apps Undergraduate Final Year Individual Project.

Appendix A

Appendix

Table A.1: MDroid+ number of mutants generated per mutation operator and application (1)

		Primitive FTPd	CPUstats	Timber	uCrop	AdvancedReciclerView	k9mail	Materialistic	MGenDatabase
Activity/Intents	ActivityNotDefined	8	3	7	3	28	30	23	29
	DifferentActivityIntentDefinition	-	-	-	-	-	-	-	-
	InvalidActivityName	8	3	7	3	28	30	23	29
	InvalidKeyInputExtra	-	-		-	-	-	-	-
	InvalidLabel	8	3	2	-	28	24	19	27
	NullIntent	-	-	-	-	-	-	-	-
	NullValueInputExtra	-	-	-	-	-	-	-	-
	WrongMainActivity	1	1	1	1	1	1	1	1
Android Programming	MissingPermissionManifest	4	1	7	3	-	12	9	-
	NotParcelable	-	-	-	-	-	-	-	-
	SDKVersion	-	-	-	-	-	-	-	-
	NullGPSLocation	-	-	-	-	-	-	-	-
	WrongStringResource	86	20	136	33	38	969	269	173
Back-End Services	NullBackEndServiceReturn	-	-	-	-	-	-	-	-
Connectivity	BluetoothAdapterAlwaysEnabled	-	-	-	-	-	-	-	-
	NullBluetoothAdapter	-	-	-	-	-	-	-	-
Data	InvalidURI	-	-	-	-	-	-	-	-
Database	ClosingNullCursor	-	-	-	-	-	-	-	-
	InvalidIndexQueryParameter	-	-	-	-	-	-	-	-
	InvalidSQLQuery	-	-	-	-	-	-	-	-
General Programming	InvalidDate	-	-	-	-	-	-	-	-
	InvalidMethodCallArgument	-	-	-	-	-	-	-	-
	NotSerializable	-	-	-	-	-	-	-	-
	NullMethodCallArgument	-	-	-	-	-	-	-	-
GUI	BuggyGuiListener	-	-	-	-	-	-	-	-
	FindViewsByIdReturnsNull	-	-	-	-	-	-	-	-
	InvalidColor	3	-	27	3	21	7	42	-
	InvalidIDFindView	-	-	-	-	-	-	-	-
	ViewComponentNotVisible	-	-	-	-	-	-	-	-
I/O	InvalidFilePath	-	-	-	-	-	-	-	-
	NullInputStream	-	-	-	-	-	-	-	-
	NullOutputStream	-	-	-	-	-	-	-	-
Non-Functional Requirements	LengthyBackEndService	-	-	-	-	-	-	-	-
	LengthyGUICreation	-	-	-	-	-	-	-	-
	LengthyGUIListener	-	-	-	-	-	-	-	-
	LongConnectionTimeOut	-	-	-	-	-	-	-	-
	OOMLargeImage	-	-	-	-	-	-	-	-
Total		118	31	187	46	144	1093	386	259

Table A.2: MDroid+ number of mutants generated per mutation operator and application (2)

		Chromadoze	GPSLogger	OpenTasks	OSRSHelper	Clementine Remote	pMetro	ShaderEditor
Activity/Intents	ActivityNotDefined	1	2	8	3	4	4	9
	DifferentActivityIntentDefinition	-	-	-	-	-	-	-
	InvalidActivityName	1	2	8	3	4	4	9
	InvalidKeyInputExtra	-	-	-	-	-	-	-
	InvalidLabel	1	2	7	1	-	4	9
	NullIntent	-	-	-	-	-	-	-
	NullValueInputExtra	-	-	-	-	-	-	-
	WrongMainActivity	-	1	1	1	1	1	1
Android Programming	MissingPermissionManifest	1	3	8	3	8	3	2
	NotParcelable	-	-	-	-	-	-	-
	SDKVersion	-	-	-	-	-	-	-
	NullGPSLocation	-	-	-	-	-	-	-
	WrongStringResource	17	134	184	145	208	52	142
Back-End Services	NullBackEndServiceReturn	-	-	-	-	-	-	-
Connectivity	BluetoothAdapterAlwaysEnabled	-	-	-	-	-	-	-
	NullBluetoothAdapter	-	-	-	-	-	-	-
Data	InvalidURI	-	-	-	-	-	-	-
Database	ClosingNullCursor	-	-	-	-	-	-	-
	InvalidIndexQueryParameter	-	-	-	-	-	-	-
	InvalidSQLQuery	-	-	-	-	-	-	-
General Programming	InvalidDate	-	-	-	-	-	-	-
	InvalidMethodCallArgument	-	-	-	-	-	-	-
	NotSerializable	-	-	-	-	-	-	-
	NullMethodCallArgument	-	-	-	-	-	-	-
GUI	BuggyGuiListener	-	-	-	-	-	-	-
	FindViewsByIdReturnsNull	-	-	-	-	-	-	-
	InvalidColor	3	18	27	31	14	3	22
	InvalidIDFindView	-	-	-	-	-	-	-
	ViewComponentNotVisible	-	-	-	-	-	-	-
I/O	InvalidFilePath	-	-	-	-	-	-	-
	NullInputStream	-	-	-	-	-	-	-
	NullOutputStream	-	-	-	-	-	-	-
Non-Functional Requirements	LengthyBackEndService	-	-	-	-	-	-	-
	LengthyGUICreation	-	-	-	-	-	-	-
	LengthyGUIListener	-	-	-	-	-	-	-
	LongConnectionTimeOut	-	-	-	-	-	-	-
	OOMLargeImage	-	-	-	-	-	-	-
Total		24	162	243	187	239	71	194

Table A.3: MDroid+ manual testing results of the background-foreground behaviour mutation operator and application (1)

		Primitive FTPd	CPUstats	Timber	uCrop	AdvancedReciclerView	k9mail	Materialistic	MGenDatabase
Activity/Intents	ActivityNotDefined	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
	DifferentActivityIntentDefinition	-	-	-	-	-	-	-	-
	InvalidActivityName	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
	InvalidKeyInputExtra	-	-		-	-	-	-	-
	InvalidLabel	Misfit	Misfit	Misfit	-	Misfit	Misfit	Misfit	Misfit
	NullIntent	-	-	-	-	-	-	-	-
	NullValueInputExtra	-	-	-	-	-	-	-	-
	WrongMainActivity	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
Android Programming	MissingPermissionManifest	Disruptive	Misfit	Disruptive	Misfit	-	Misfit	Misfit	-
	NotParcelable	-	-	-	-	-	-	-	-
	SDKVersion	-	-	-	-	-	-	-	-
	NullGPSLocation	-	-	-	-	-	-	-	-
	WrongStringResource	Misfit	Misfit	Misfit	Misfit	Misfit	Misfit	Misfit	Misfit
Back-End Services	NullBackEndServiceReturn	-	-	-	-	-	-	-	-
Connectivity	BluetoothAdapterAlwaysEnabled	-	-	-	-	-	-	-	-
	NullBluetoothAdapter	-	-	-	-	-	-	-	-
Data	InvalidURI	-	-	-	-	-	-	-	-
Database	ClosingNullCursor	-	-	-	-	-	-	-	-
	InvalidIndexQueryParameter	-	-	-	-	-	-	-	-
	InvalidSQLQuery	-	-	-	-	-	-	-	-
General Programming	InvalidDate	-	-	-	-	-	-	-	-
	InvalidMethodCallArgument	-	-	-	-	-	-	-	-
	NotSerializable	-	-	-	-	-	-	-	-
	NullMethodCallArgument	-	-	-	-	-	-	-	-
GUI	BuggyGuiListener	-	-	-	-	-	-	-	-
	FindViewsByIdReturnsNull	-	-	-	-	-	-	-	-
	InvalidColor	Disruptive	-	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
	InvalidIDFindView	-	-	-	-	-	-	-	-
	ViewComponentNotVisible	-	-	-	-	-	-	-	-
I/O	InvalidFilePath	-	-	-	-	-	-	-	-
	NullInputStream	-	-	-	-	-	-	-	-
	NullOutputStream	-	-	-	-	-	-	-	-
Non-Functional Requirements	LengthyBackEndService	-	-	-	-	-	-	-	-
	LengthyGUICreation	-	-	-	-	-	-	-	-
	LengthyGUIListener	-	-	-	-	-	-	-	-
	LongConnectionTimeOut	-	-	-	-	-	-	-	-
	OOMLargeImage	-	-	-	-	-	-	-	-

Table A.4: MDroid+ manual testing results of the background-foreground behaviour mutation operator and application (2)

		Chromadoze	GPSLogger	OpenTasks	OSRSHelper	Clementine Remote	pMetro	ShaderEditor
Activity/Intents	ActivityNotDefined	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
	DifferentActivityIntentDefinition	-	-	-	-	-	-	-
	InvalidActivityName	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
	InvalidKeyInputExtra	-	-	-	-	-	-	-
	InvalidLabel	Misfit	Misfit	Misfit	Misfit	Misfit	Misfit	Misfit
	NullIntent	-	-	-	-	-	-	-
	NullValueInputExtra	-	-	-	-	-	-	-
	WrongMainActivity	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
Android Programming	MissingPermissionManifest	Misfit	Misfit	Misfit	Disruptive	Misfit	Misfit	Misfit
	NotParcelable	-	-	-	-	-	-	-
	SDKVersion	-	-	-	-	-	-	-
	NullGPSLocation	-	-	-	-	-	-	-
	WrongStringResource	Misfit	Misfit	Misfit	Misfit	Misfit	Misfit	Misfit
Back-End Services	NullBackEndServiceReturn	-	-	-	-	-	-	-
Connectivity	BluetoothAdapterAlwaysEnabled	-	-	-	-	-	-	-
	NullBluetoothAdapter	-	-	-	-	-	-	-
Data	InvalidURI	-	-	-	-	-	-	-
Database	ClosingNullCursor	-	-	-	-	-	-	-
	InvalidIndexQueryParameter	-	-	-	-	-	-	-
	InvalidSQLQuery	-	-	-	-	-	-	-
General Programming	InvalidDate	-	-	-	-	-	-	-
	InvalidMethodCallArgument	-	-	-	-	-	-	-
	NotSerializable	-	-	-	-	-	-	-
	NullMethodCallArgument	-	-	-	-	-	-	-
GUI	BuggyGuiListener	-	-	-	-	-	-	-
	FindViewsByIdReturnsNull	-	-	-	-	-	-	-
	InvalidColor	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive	Disruptive
	InvalidIDFindView	-	-	-	-	-	-	-
	ViewComponentNotVisible	-	-	-	-	-	-	-
I/O	InvalidFilePath	-	-	-	-	-	-	-
	NullInputStream	-	-	-	-	-	-	-
	NullOutputStream	-	-	-	-	-	-	-
Non-Functional Requirements	LengthyBackEndService	-	-	-	-	-	-	-
	LengthyGUICreation	-	-	-	-	-	-	-
	LengthyGUIListener	-	-	-	-	-	-	-
	LongConnectionTimeOut	-	-	-	-	-	-	-
	OOMLargeImage	-	-	-	-	-	-	-

Table A.5: Number of mutant applications generate by the tool developed per mutation operator and application

Application	**onSaveInstanceState**	**EditText**	**Spinner**	**Intent**
Advanced RecyclerView Examples	5	0	0	28
Amaze File Manager	7	0	0	8
aMetro	2	0	0	6
And Bible	0	2	4	27
AnkiDroid Flashcards	5	1	3	20
AntennaPod	3	3	1	20
Anuto TD	0	0	0	4
BankDroid	2	0	0	14
Blokish	0	0	0	4
Chroma Doze	0	0	1	0
Clementine Remote	0	1	0	4
ConnectBot	0	2	1	11
CPU Stats	0	0	0	3
CycleStreet	0	0	0	0
Debatekeeper	0	0	0	4
Forkhub	6	0	0	29
Glucosio	0	0	0	16
GPS Logger	0	0	0	2
k-9 mail	10	7	3	2
LibreTorrent	16	0	1	8
Lottie	0	0	0	0
Material Dialogs Library Demo	0	0	0	2
Materialistic - Hacker News	13	3	0	23
MHGen Database	1	1	0	28
MIFARE Classic Tool	1	5	0	17
My Diary	0	2	1	12
Omni Notes	3	1	1	12
OneBusAway	10	3	1	27
OpenBikeSharing	2	0	0	5
OpenTasks	3	3	1	7
OSRS Helper	1	0	0	3
Password Store	0	0	0	0
Photo Affix	0	0	0	2
pMetro	0	0	0	3
Primitive FTPd	0	0	0	9
qBittorent Controller	4	0	0	7
RedReader	7	1	0	17
RGB Tool	0	0	0	6
Shader Editor	0	0	1	5
ShutUp!	0	0	0	6
SwiftNotes	1	1	0	2
Timber	0	0	0	9
uCrop	0	0	0	0
Unit Converter Ultimate	0	0	0	2
Vlille Checker	0	0	0	22
Web Opac: 1000+ libraries	6	0	0	12
Weechat Android	0	0	0	4
WiFi Automatic	0	0	0	0
WifiAnalyser	0	0	0	4
WiGLE Wifi Wardriving	4	0	0	10
Total	112	36	19	466

Table A.6: Datasets applications general information and results of Manual Testing and *iMPAct tool* testing of the background pattern (1)

Application	**Category**	**Rating**	**Instalations**	**Manual Backgorund-Foreground Behaviour Testing**	**iMPAcT tool Background Pattern Testing**
Advanced RecyclerView Examples	Library and Demo	4.8	10 000 - 50 000	Correct	Correct
Amaze File Manager	Tools	4.3	500 000 - 1 000 000	Correct	Correct
aMetro	Maps and Navigation	4.4	100 000 - 500 000	Correct	Correct
And Bible	Books and Reference	4.6	100 000 - 500 000	Correct	Correct
AnkiDroid Flashcards	Education	4.5	1 000 000 - 5 000 000	Correct	Correct
AntennaPod	Video Players & Editors	4.6	100 000 - 500 000	Incorrect	Correct
Anuto TD	Strategy	4.3	10 000 - 50 000	Correct	Correct
BankDroid	Finance	4.1	100 000 - 500 000	Correct	Correct
Blokish	Puzzle	4.3	100 000 - 500 000	Correct	Correct
Chroma Doze	Music and Audio	4.5	50 000 - 100 000	Correct	Correct
Clementine Remote	Music and Audio	4.5	100 000 - 500 000	Correct	Correct
ConnectBot	Communication	4.6	1 000 000 - 5 000 000	Correct	Correct
CPU Stats	Tools	4.5	100 000 - 500 000	Correct	Correct
CycleStreet	Travel and Local	4.7	10 000 - 50 000	Correct	Correct
Debatekeeper	Tools	4.6	10 000 - 50 000	Correct	Correct
Forkhub	Productivity	4.4	10 000 - 50 000	Correct	Correct
Glucosio	Medical	4.2	10 000 - 50 000	Correct	Correct
GPS Logger	Travel and Local	4.2	500 000 - 1 000 000	Correct	Correct
k-9 mail	Communication	4.2	5 000 000 - 10 000 000	Correct	Correct
LibreTorrent	Video Players & Editors	4.3	10 000 - 50 000	Correct	Correct
Lottie	Library and Demo	4.7	10 000 - 50 000	Incorrect	Correct
Material Dialogs Library Demo	Library and Demo	4.8	10 000 - 50 000	Correct	Correct
Materialistic - Hacker News	News and Magazine	4.8	50 000 - 100 000	Correct	Correct
MHGen Database	Books and Reference	4.8	100 000 - 500 000	Correct	Correct
MIFARE Classic Tool	Tools	4.3	100 000 - 500 000	Incorrect	Correct
My Diary	Lifestyle	4.8	100 000 - 500 000	Correct	Correct
Omni Notes	Productivity	4.4	100 000 - 500 000	Correct	Correct
OneBusAway	Maps and Navigation	4.3	500 000 - 1 000 000	Correct	Correct
OpenBikeSharing	Travel and Local	4.2	10 000 - 50 000	Incorrect	Correct

Table A.7: Datasets applications general information and results of Manual Testing and *iMPAct tool* testing of the background pattern (2)

Application	**Category**	**Rating**	**Instalations**	**Manual Backgorund-Foreground Behaviour Testing**	**iMPAcT tool Background Pattern Testing**
OpenTasks	Productivity	4.2	100 000 - 500 000	Correct	Correct
OSRS Helper	Tools	4.4	10 000 - 50 000	Correct	Correct
Password Store	Productivity	4.6	10 000 - 50 000	Correct	Correct
Photo Affix	Tools	4.5	10 000 - 50 000	Correct	Correct
pMetro	Travel and Local	4.1	10 000 - 50 000	Correct	Correct
Primitive FTPd	Tools	4.5	10 000 - 50 000	Correct	Correct
qBittorent Controller	Tools	3.8	100 000 - 500 000	Correct	Correct
RedReader	News and Magazines	4.6	50 000 - 100 000	Correct	Correct
RGB Tool	Tools	4.3	10 000 - 50 000	Correct	Correct
Shader Editor	Tools	4.7	10 000 - 50 000	Correct	Correct
ShutUp!	Productivity	4.4	10 000 - 50 000	Correct	Correct
SwiftNotes	Productivity	4.3	10 000 - 50 000	Correct	Correct
Timber	Music and Audio	4.3	100 000 - 500 000	Correct	Correct
uCrop	Photography	4.0	10 000 - 50 000	Correct	Correct
Unit Converter Ultimate	Tools	4.5	1 000 000 - 5 000 000	Correct	Correct
Vlille Checker	Maps and Navigation	4.7	10 000 - 50 000	Correct	Correct
Web Opac: 1000+ libraries	Education	4.3	50 000 - 100 000	Correct	Correct
Weechat Android	Communication	4.3	50 000 - 100 000	Correct	Correct
WiFi Automatic	Tools	4.1	1 000 000 - 5 000 000	Correct	Correct
WifiAnalyser	Tools	4.4	1 000 000 - 5 000 000	Correct	Correct
WiGLE Wifi Wardriving	Tools	4.2	500 000 - 1 000 000	Incorrect	Correct